HZ BOOKS
华章科技

超过15年IT行业从业经验的Spring Boot专家撰写，系统讲解Spring Boot的各项关键技术
结合实际生产环境讲解Spring Boot分布式应用开发及高性能服务平台搭建
深入剖析Spring Boot核心功能的源码实现

深入实践 Spring Boot

Deep into Spring Boot

陈韶健 著

机械工业出版社
China Machine Press

内容简介

本书是Spring Boot领域的经典著作，从技术、实践和原理3个维度对Spring Boot进行了系统且深入的讲解。作者是Spring Boot领域的资深专家，有超过15年的IT行业从业经验。

技术维度

第一部分（1~5章），针对性地介绍了Spring Boot入门、数据库的使用和访问性能提升、界面设计、安全设计等重要技术知识，以实用性为主，旨在帮助读者快速掌握Spring Boot开发方法和精髓，尽快融入生产实践中。

实践维度

第二部分（6~9章），用生产环境中的实际案例讲解了如何使用Spring Boot开发分布式应用和云应用，以及如何用微服务构建高可用的服务平台，实践性极强。

原理维度

第三部分（10~12章），从源码层面着重分析了Spring Boot的程序加载、自动配置、数据管理、Spring Cloud的配置管理、发现服务和负载均衡服务等核心功能的实现原理，旨在帮助读者能更深刻地理解Spring Boot开发，掌握其精髓。

深入实践 Spring Boot

Deep into Spring Boot

陈韶健 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入实践 Spring Boot / 陈韶健著 . —北京：机械工业出版社，2016.10

ISBN 978-7-111-55088-4

I. 深… II. 陈… III. JAVA 语言 – 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2016）第 244089 号

深入实践 Spring Boot

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺　　责任校对：殷 虹

印　　刷：北京文昌阁彩色印刷有限责任公司　　版　　次：2016 年 11 月第 1 版第 1 次印刷

开　　本：186mm × 240mm 1/16　　印　　张：17.25

书　　号：ISBN 978-7-111-55088-4　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

Preface 前 言

Spring Boot 作为 Java 编程语言的一个全新开发框架，在国内外才刚刚兴起，还未得到普及使用。相比于以往的一些开发框架，Spring Boot 不但使用更加简单，而且功能更加丰富，性能更加稳定而健壮。使用 Spring Boot 开发框架，不仅能提高开发速度，增强生产效率，从某种意义上，可以说是解放了程序员的劳动，而且一种新技术的使用，更能增强系统的稳定性和扩展系统的性能指标。本书就是本着提高开发效率，增强系统性能，促进新技术的普及使用这一目的而写的。

Spring Boot 是在 Spring 框架基础上创建的一个全新框架，其设计目的是简化 Spring 应用的搭建和开发过程，它不但具有 Spring 的所有优秀特性，而且具有如下显著的特点：

- 为 Spring 开发提供更加简单的使用和快速开发的技巧。
- 具有开箱即用的默认配置功能，能根据项目依赖自动配置。
- 具有功能更加强大的服务体系，包括嵌入式服务、安全、性能指标、健康检查等。
- 绝对没有代码生成，可以不再需要 XML 配置，即可让应用更加轻巧和灵活。

Spring Boot 对于一些第三方技术的使用，提供了非常完美的整合，使你在简单的使用中，不知不觉运用了非常高级和先进的技术。

虽然 Spring Boot 具有这么多优秀的特性，但它使用起来并不复杂，而且非常简单，所以不管是 Java 程序开发初学者，还是经验丰富的开发人员，使用 Spring Boot 都是一个理想的选择。

Spring Boot 发展迅速，自从 2014 年 4 月发布了 1.0.0 版本，至今已经发布了 1.4.0 正式版。现在，Spring Boot 正在不同的角落中悄然兴起，估计用不了多久，它将成为

Java 开发的又一个热潮，为众多 Java 开发者追捧。

本书将以一些非常切合生产实际的应用实例，带你一起使用 Spring Boot 框架，开始一段愉快的快速开发和探索之旅。

关于本书

本书以丰富的实例，介绍了如何使用 Spring Boot 开发框架进行基础应用和分布式应用等方面的开发，并且介绍了如何使用 Spring Boot 开发的应用搭建一个高性能的服务平台，同时还对 Spring Boot 的一些核心功能的源代码进行了分析，从而加深对 Spring Boot 的理解。书中对从最基本的入门知识，到数据库的使用，以及界面设计、安全设计等领域都做了详细的介绍和探讨，并在分布式应用系统领域，以平台级应用系统的实例，介绍了如何创建和使用 SSO 管理系统、分布式文件系统，如何使用 Spring Cloud 进行云应用方面的开发，以及如何使用 Docker 发布和构建高可用的分布式系统服务平台。同时，对 Spring Boot 的程序加载、自动配置、数据管理，和 Spring Cloud 的配置管理、发现服务和负载均衡服务等核心功能的源代码做了深入剖析，这样在认识其实现原理的基础上，能更好地使用其相应的功能。

全书分为三个部分：第一部分（第 1 ~ 5 章）介绍基础应用方面的开发，包含简单入门知识、数据库使用、界面设计和安全设计等内容；第二部分（第 6 ~ 9 章）介绍了 Spring Boot 在分布式系统开发和云应用开发等方面的应用以及使用微服务构建高可用的服务平台；第三部分（第 10 ~ 12 章）对 Spring Boot 的程序加载、自动配置和数据管理的实现原理，以及 Spring Cloud 的配置管理、发现服务和负载均衡服务等实现原理进行了深入的剖析。

本书章节编排

第 1 章为 Spring Boot 入门，介绍开发环境的搭建和开发工具的选择及安装配置，并使用一个非常简单的实例，说明如何轻易地使用 Spring Boot 开发框架。

第 2 章使用 Spring Boot 框架演示了以不同于以往的方式，以及如何轻易地使用数据库，并实际演示使用 MySQL、MongoDB、Redis 和 Neo4j 等数据库。

第 3 章使用 Thymeleaf 模板结合一些流行的 JavaScript 插件，介绍了使用 Spring Boot 进行界面设计的方法和技巧。

第 4 章对使用 Spring Boot 提高传统关系型数据库的性能方面做了一些探讨和尝试，

并扩展了使用 JPA 资源库的功能。

第 5 章介绍了如何使用 Spring Boot 结合 Spring Security 进行安全设计，包括登录认证和角色管理、权限管理等内容。

第 6 章介绍如何使用 Spring Security 结合 OAuth2 进行 SSO（Single Sign On）的设计，并演示如何在分布式应用系统中使用认证授权和安全管理的功能。

第 7 章介绍如何使用 Spring Boot 框架结合分布式文件系统 FastDFS，并使用定制方式和富文本编辑器的方式演示了使用图片上传和建立本地图片库的方法。

第 8 章介绍云应用开发，包括配置管理、发现服务和监控服务的使用，以及如何使用动态路由和断路器的功能，创建高可用的微服务应用。

第 9 章介绍如何使用 Docker 引擎和 docker-compose 工具来发布应用和管理服务，以及如何构建一个高性能的服务平台和怎样使用 Docker 实施负载均衡。

第 10 章分析了 Spring Boot 的应用程序加载和自动配置原理，以及如何以改造加载配置的方式来提高应用的性能。

第 11 章分析了 Spring Boot 使用数据库的实现原理，并演示怎样利用一些技术手段提高和扩展访问数据库的功能。

第 12 章简要分析了微服务中配置管理、发现服务和负载均衡服务的实现原理和部分核心源代码，并使用一个实例说明配置管理中分布式消息的实现机制和原理。

附录 A ~ 附录 D 介绍了 Neo4j、MongoDB、Redis、RabbitMQ 等服务器的安装、配置和基本使用方法。

读者对象

本书适于所有 Java 编程语言开发人员，所有对 Spring Boot 感兴趣并希望使用 Spring Boot 开发框架进行开发的人员，已经使用过 Spring Boot 框架但希望更好地使用 Spring Boot 的开发人员，以及系统设计师、架构师等设计人员。同时，本书对运维人员和 DBA 等也具有一定的参考价值。

实例代码

本书的实例代码可以通过 https://github.com/chenfromsz?tab=repositories 查看和下载，推荐根据每章的提示使用 IntelliJ IDEA 通过 GitHub 检出各章的实例工程，这样可以保留原来工程的配置，并且能够直接使用。

反馈与勘误

读者如有反馈意见可以通过 https://github.com/chenfromsz/correct/issues 发起新话题与作者进行交互，在这也可能会发布一些勘误信息，以便纠正不可避免的错误。

致谢

首先，非常感谢华阳信通公司，虽然本书的编写过程大都在业余时间完成，但是公司强大的平台给本书的实例提供了更加方便的分享、验证和测试条件。同时在本书的编写过程中，也得到了我们的开发团队和众多朋友的大力支持和帮助，在此表示衷心的感谢！最后感谢华章公司的杨福川和李艺，他们在本书编辑的过程中，提出了一些宝贵而有益的建议，并为本书的出版做了许多工作。

由于时间仓促和水平有限，书中难免出现一些纰漏或不正确的地方，敬请大家批评指正！

Contents 目　录

第二部分 分布式应用开发

第一部分 *Part 1*

基础应用开发

这一部分从搭建开发环境，简单入门，到使用数据库、界面设计、安全管理等一系列内容，介绍了使用 Spring Boot 框架进行基础应用开发的方法。

第 1 章介绍了开发环境的搭建和开发工具的选择和安装，并以一个非常简单的实例，演示了如何使用 Spring Boot 框架创建工程和发布应用。

第 2 章介绍了如何用 Spring Boot 特有的方式，使用当前流行的数据库：MySQL、Redis、MongoDB、Neo4j 等。

第 3 章介绍如何使用 Thymeleaf 模板结合一些流行的 JavaScript 插件，设计应用界面。

第 4 章使用 Druid 数据库连接池和 Redis 做缓存来尝试提升关系型数据库的访问性能，并扩展了 JPA 的资源库功能。

第 5 章在 Spring Boot 中使用 Spring Security 为应用系统进行安全设计，实现了登录认证和权限管理方面的功能。

第 1 章 Chapter 1

Spring Boot 入门

在使用 Spring Boot 框架进行各种开发体验之前，要先配置好开发环境。首先安装 JDK，然后选择一个开发工具，如 Eclipse IDE 和 IntelliJ IDEA（以下简称 IDEA）都是不错的选择。对于开发工具的选择，本书极力推荐使用 IDEA，因为它为 Spring Boot 提供了许多更好和更贴切的支持，本书的实例都是使用 IDEA 创建的。同时，还需要安装 Apache Maven 和 Git 客户端。所有这些都准备好之后，我们就能开始使用 Spring Boot 了。

1.1 配置开发环境

下面的开发环境配置主要以使用 Windows 操作系统为例，如果你使用的是其他操作系统，请对照其相关配置进行操作。

1.1.1 安装 JDK

JDK（Java SE Development Kit）需要 1.8 及以上版本，可以从 Java 的官网 http://www.oracle.com/technetwork/java/javase/downloads/index.html 下载安装包。如果访问官网速度慢的话，也可以通过百度搜索 JDK，然后在百度软件中心下载符合你的 Windows 版本和配置的 JDK1.8 安装包。

安装完成后，配置环境变量 JAVA_HOME，例如，使用路径 D:\Program Files\Java\jdk1.8.0_25（如果你安装的是这个目录的话）。JAVA_HOME 配置好之后，将 %JAVA_HOME%\bin 加入系统的环境变量 path 中。完成后，打开一个命令行窗口，输入命令 java–version，如果能正确输出版本号则说明安装成功了。输出版本的信息如下：

```
C:\Users\Alan>java-version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

1.1.2 安装 InterlliJ IDEA

IDEA 需要 14.0 以上的版本，可以从其官网 http://www.jetbrains.com/ 下载免费版，本书的实例是使用 IDEA14.1.15 版本开发的。IDEA 已经包含 Maven 插件，版本是 3.0.5，这已经能够适用我们开发的要求。安装完成后，打开 IDEA，将显示如图 1-1 所示的欢迎界面，在这里可以看到 IDEA 的版本号。

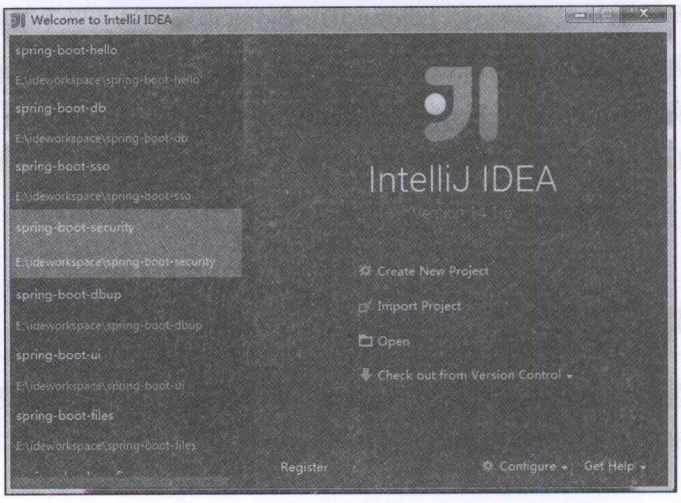


图 1-1 InterlliJ IDEA 欢迎界面

1.1.3 安装 Apache Maven

为了能够在命令行窗口中使用 Maven 来管理工程，可以安装一个 Maven 管理工具。通过 Maven 的官网 http://maven.apache.org/download.cgi 下载 3.0.5 以上的版本，下

载完后解压缩即可，例如，解压到 D: 盘上是不错的做法，然后将 Maven 的安装路径（如 D:\apache-maven-3.2.3\bin）也加入 Windows 的环境变量 path 中。安装完成后，在命令行窗口中执行指令：mvn–v，将输出如下的版本信息以及系统的一些环境信息。

```
C:\Users\Alan>mvn-v
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-
12T04:58:1
0+08:00)
Maven home: D:\apache-maven-3.2.3\bin\..
Java version: 1.8.0_25, vendor: Oracle Corporation
Java home: D:\Program Files\Java\jdk1.8.0_25\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
```

建议更改 IDEA 中 Maven 资源库的存放路径，可以先在 Maven 安装路径中创建一个资源库目录，如 repository。然后打开 Maven 的配置文件，即安装目录 conf 中的 settings.xml，找到下列代码，将路径更改为 repository 所在的位置，并保存在注释符下面。

例如找到下列代码行：

```
<localRepository>/path/to/local/repo</localRepository>
```

复制出来改为如下所示：

```
<localRepository>D:\apache-maven-3.2.3\repository</localRepository>
```

改好后可以拷贝一份 settings.xml 放置在 ${user.home}/.m2/ 下面，这样做可以不用修改 IDEA 的 Maven 这个配置。在图 1-2 所示的 Maven 配置界面中，User Settings File 保持了默认位置，Local Repository 使用了上面设置的路径 D:\apache-maven-3.2.3\repository，而 Maven 程序还是使用了 IDEA 自带的版本。

1.1.4　安装 Git 客户端

由于本书的实例工程都存放在 GitHub（https://github.com/）中，所以还需要在 GitHub 中免费注册一个用户（可以通过 E-mail 直接注册免费用户），以方便在 IDEA 中从 GitHub 检出本书的实例工程。当然，如果不想注册，通过普通下载的方法也能取得实例工程的源代码。GitHub 是世界级的代码库服务器，如果你愿意，也可以将它作为你的代码库服务器，在这里还可以搜索到全世界的开发者分享出来的源程序。图 1-3 是打开 GitHub 的首页。

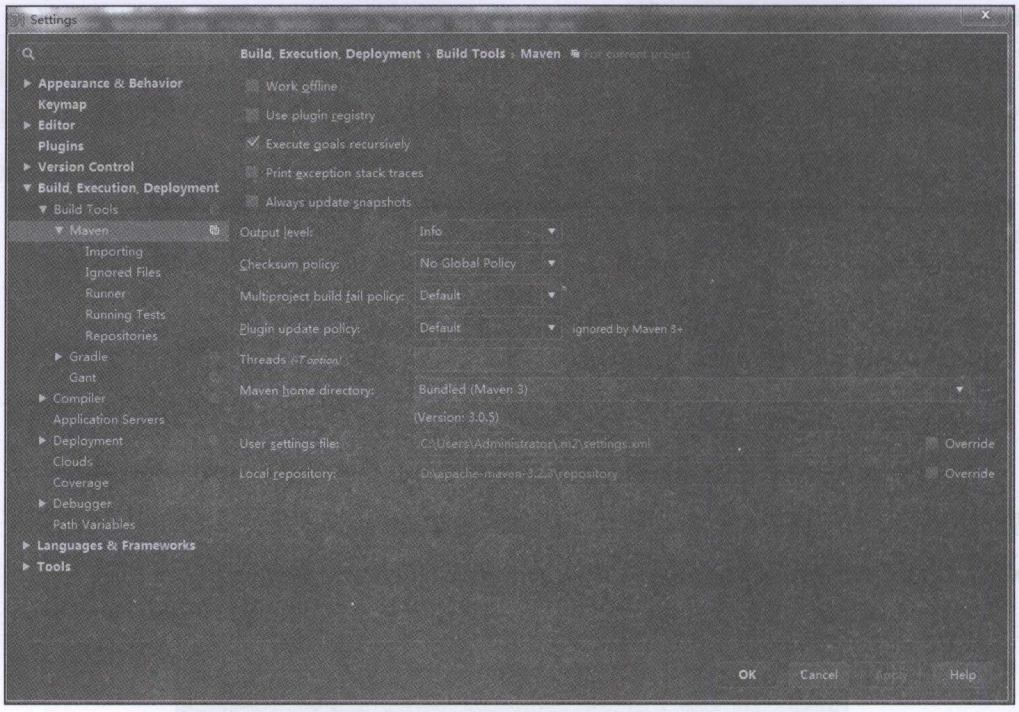


图 1-2 Maven 设置

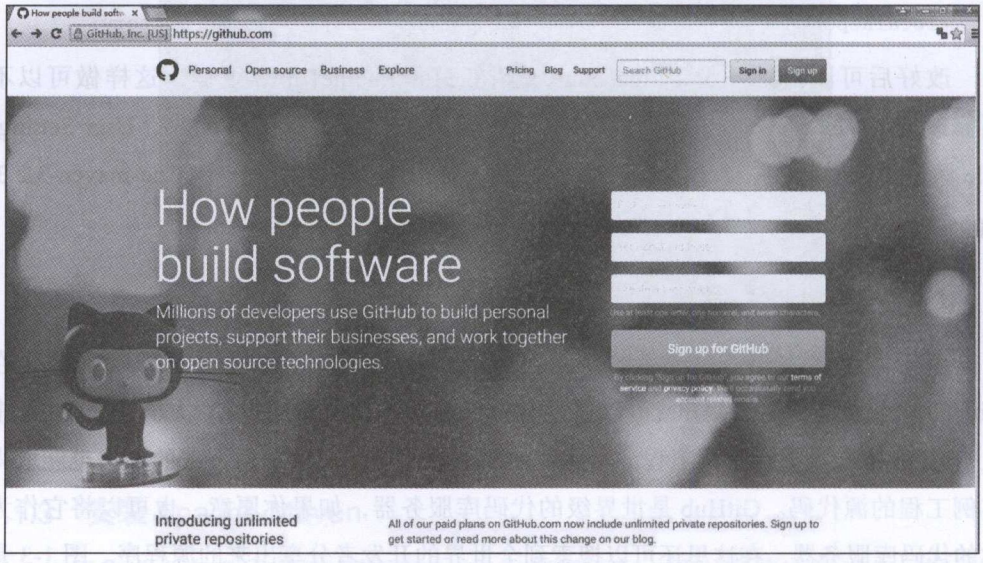


图 1-3 GitHub 首页

IDEA 还需要 Git 客户端程序的支持。可以从其官网 https://git-scm.com/download/

下载 Git 客户端安装包。安装非常简单，按提示单击“下一步”并选择好安装路径即可。安装完成后，在 Windows 的资源管理器中，单击鼠标右键弹出的菜单中将会多出如下几个选择菜单：

```
Git Init Here
Git Gui
Git Bash
```

其中 Git Bash 是一个带有 UNIX 指令的命令行窗口，在这里可以执行一些 Git 指令，用来提交或者检出项目。

在 IDEA 中对 Git 的设置，只要指定 git.exe 执行文件的位置即可。图 1-4 是 IDEA 中 Git 客户端的配置，其中 Git 的路径被设置在 D:\Program Files\Git\bin\git.exe 中，这主要由安装 Git 客户端的位置而定。

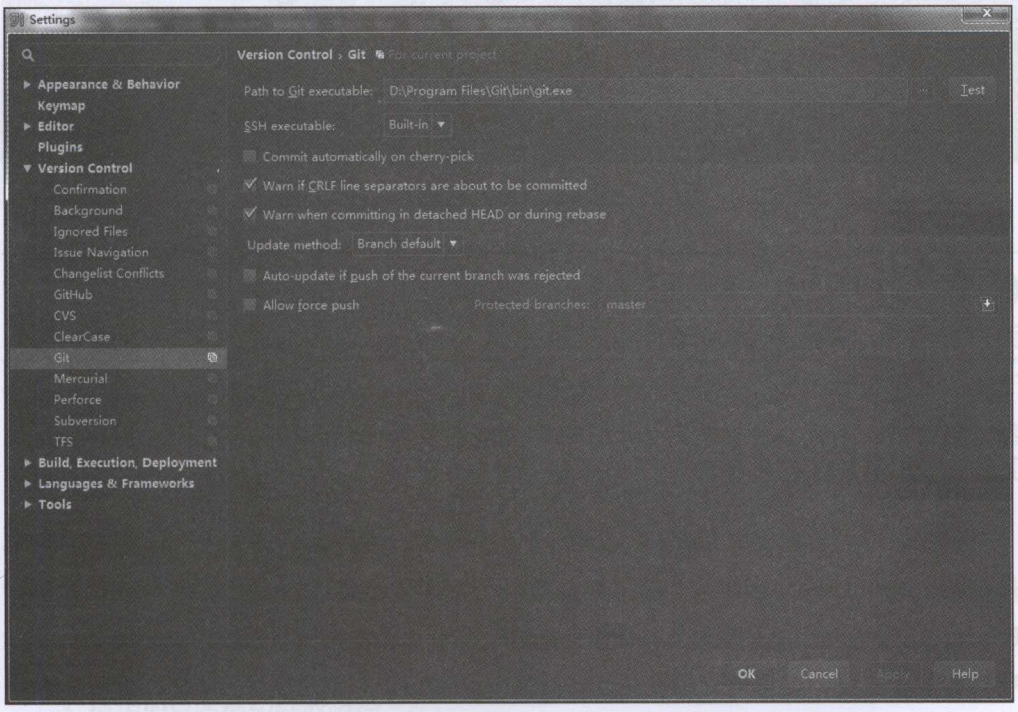


图 1-4　Git 设置

如果已经在 GitHub 中注册了用户，即可以打开如图 1-5 所示的 GitHub 配置，输入用户名和密码，然后单击 Test 按钮，如果设置正确的话将会返回连接成功的提示。

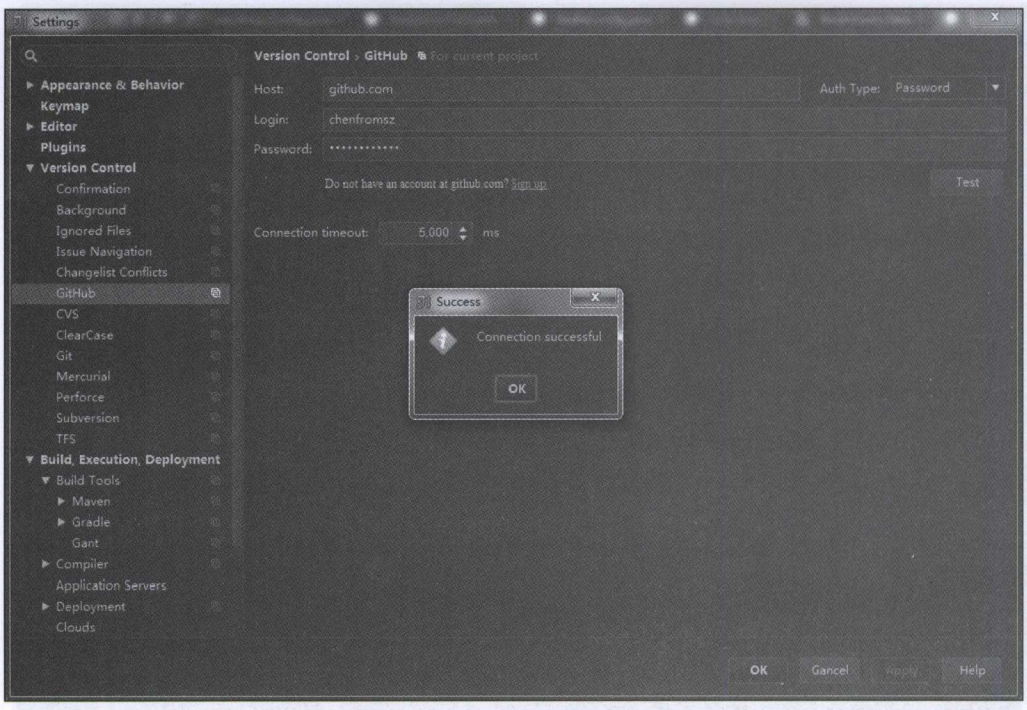


图 1-5 GitHub 配置

上面 IDEA 的一些设置界面都可以单击工具栏上的 Settings 按钮打开，打开 File 菜单，选择 Settings 同样也可以打开。

1.2 创建项目工程

现在，可以尝试使用 IDEA 来创建一个项目工程。如果是第一次打开 IDEA，可以选择 Create New Project 创建一个新工程。如果已经打开了 IDEA，在 File 菜单中选择 New Project，也能打开 New Project 对话框，如图 1-6 所示。使用 IDEA 创建一个 Spring Boot 项目有很多方法，这里只介绍使用 Maven 和 Spring Initializr 这两种方法来创建一个新项目。一般使用 Maven 来新建一个项目，因为这样更容易按我们的要求配置一个项目。

1.2.1 使用 Maven 新建项目

使用 Maven 新建一个项目主要有以下三个步骤。

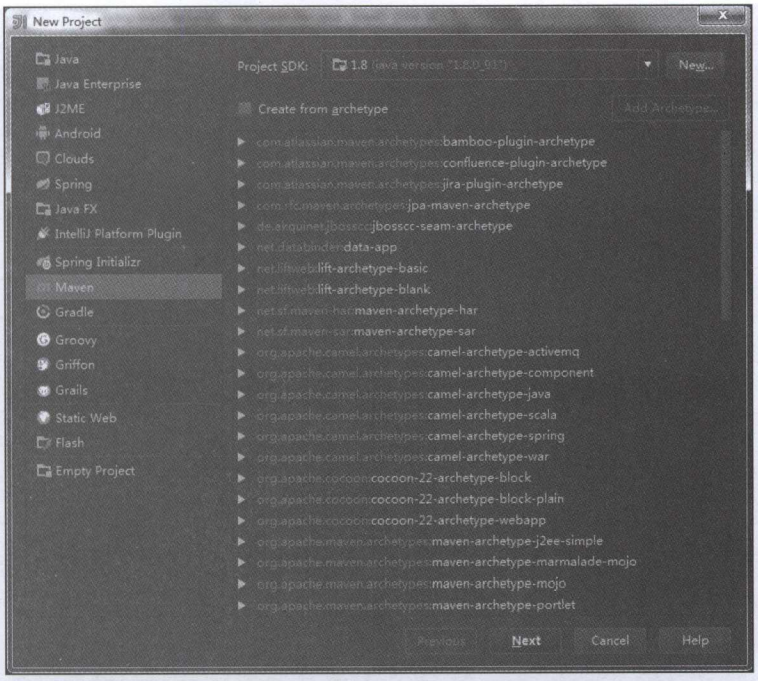


图 1-6　新建一个 Maven 项目

1. 选择项目类型

在图 1-6 中的 Project SDK 下拉列表框中选择前面安装的 Java 1.8，如果下拉列表框中不存在 Java 1.8，可以单击 New 按钮，找到安装 Java 的位置，选择它。然后在左面侧边栏的项目类型中，选择 Maven 项目，即可使用 Maven 作为项目的管理工具。至于 Maven 中的 archetype，因为我们并不打算使用其中任何一种类型，所以不用勾选，然后单击 Next 进入下一步。

2. 输入 GroupId 和 ArtifactId

在 GroupId 输入框中输入“springboot.example”，在 ArtifactId 输入框中输入“spring-boot-hello”，Version 输入框中保持默认值，如图 1-7 所示，单击 Next 进入下一步。

3. 指定项目名称和存放路径

在 Project location 编辑框中选择和更改存放路径，在 Project name 输入框中输入与 ArtifactId 相同的项目名称：“spring-boot-hello”，如图 1-8 所示。

单击 Finish，完成项目创建，这样将在当前窗口中打开一个新项目，如图 1-9 所

示。其中，在工程根目录中生成了一个 pom.xml，即 Maven 的项目对象模型（Project Object Model），并生成了源代码目录 java、资源目录 resources 和测试目录 test 等，即生成了一个项目的一些初始配置和目录结构。

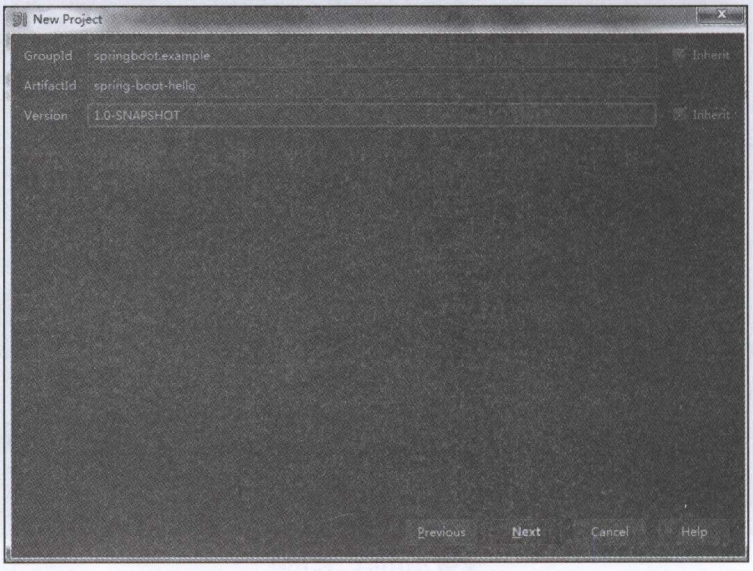


图 1-7 输入 GroupId 和 ArtifactId

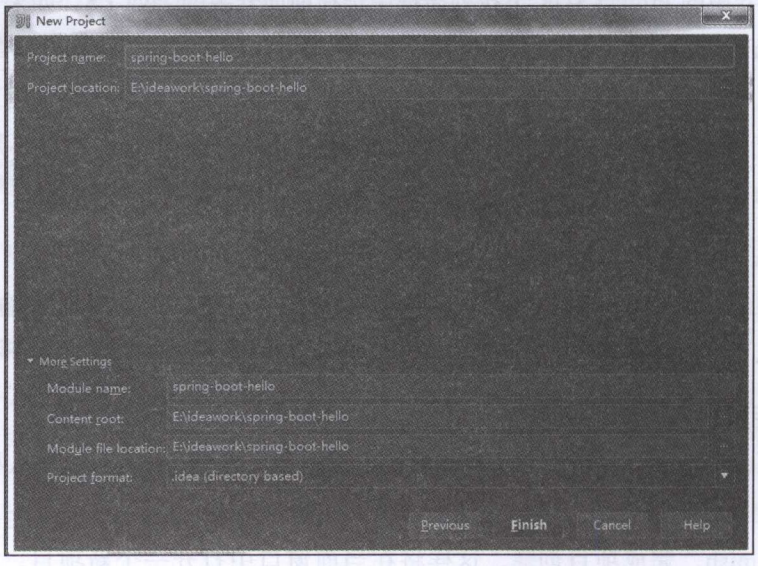


图 1-8 指定项目名称和存放路径

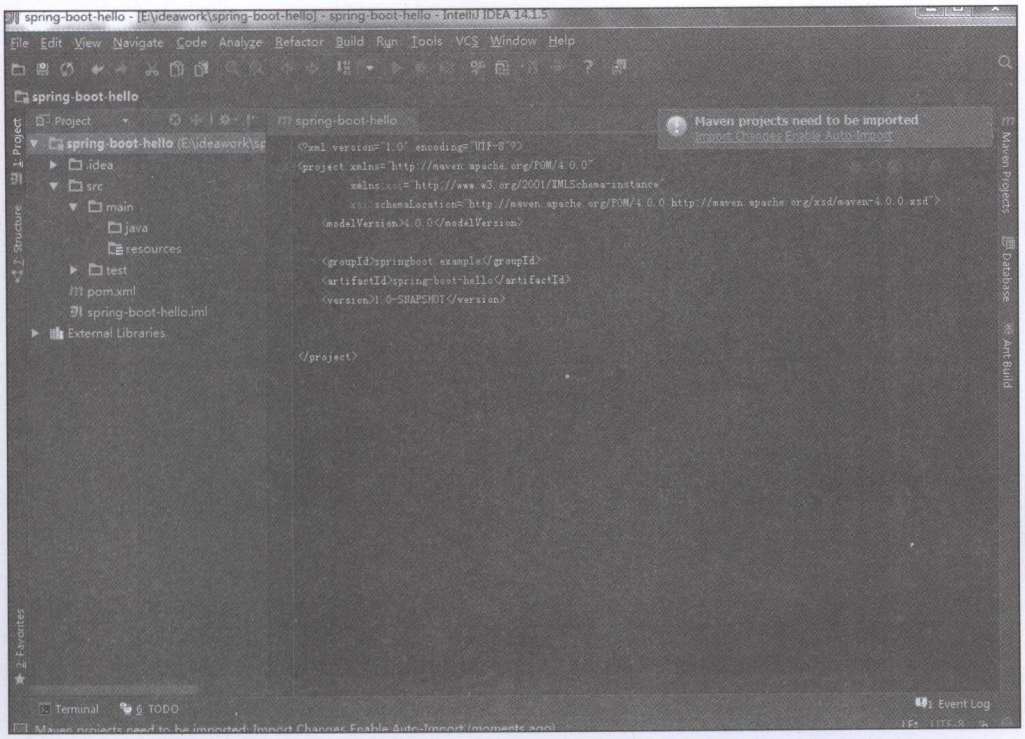


图 1-9　初始创建的项目

下一节将使用这个项目工程来创建第一个使用 Spring Boot 开发框架的应用实例。

1.2.2　使用 Spring Initializr 新建项目

新建一个 Spring Boot 项目，也可以使用 Spring Initializr 的方式，这种方式很简单，如图 1-10 所示。注意 Initializr Service URL 为 https://start.spring.io，这将会连接网络，以查询 Spring Boot 的当前可用版本和组件列表。使用这种方式新建项目大体上也需要三个步骤。

1. 选择类型

可以使用默认选项，注意 Type 为 Maven Project，Java Version 为 1.8，Packaging 为 Jar，如图 1-11 所示。单击 Next 进入下一步。

2. 选择 Spring Boot 版本和组件

选择 Spring Boot 版本和 Spring Boot 组件，例如，在 Spring Boot Version 中选择 1.3.5，并勾选 Web 项目组件，如图 1-12 所示，然后单击 Next 进入下一步。

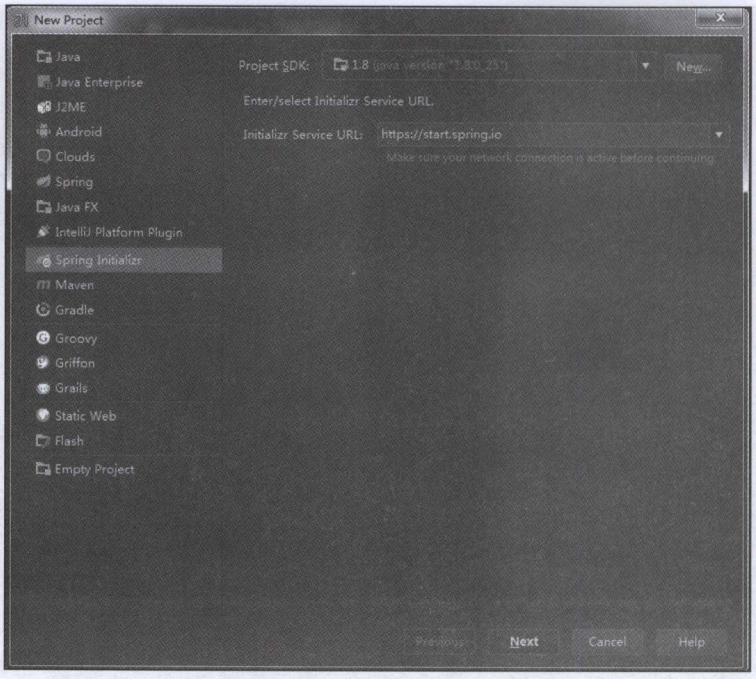


图 1-10 新建一个 Spring Boot 项目

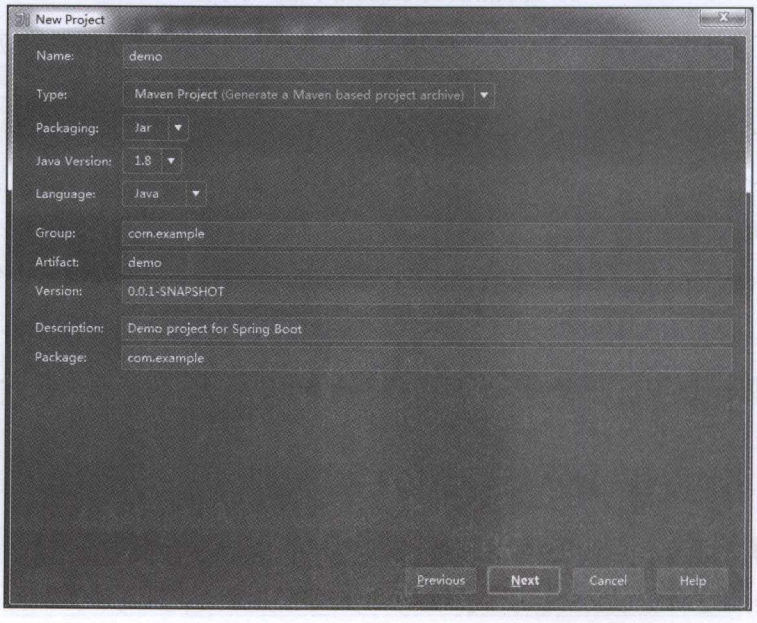


图 1-11 选择项目类型

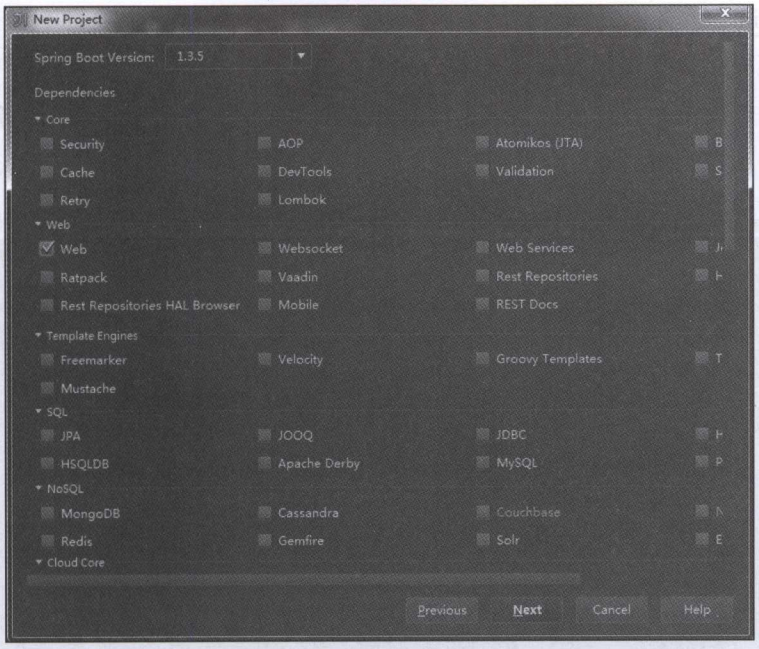


图 1-12　选择版本和组件

3. 输入项目名称

选择存放路径后输入项目名称，如图 1-13 所示，这里使用 demo 作为项目的名称。

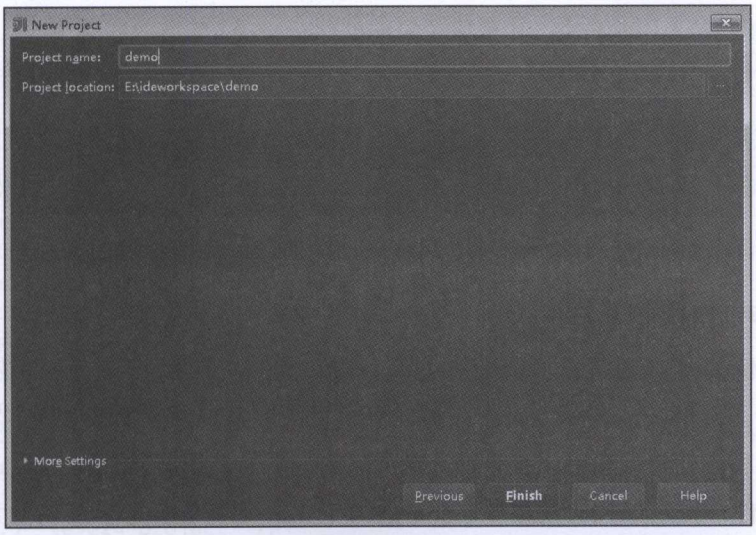


图 1-13　输入项目名称

单击 Finish，将创建一个初始化项目，如图 1-14 所示。这个项目不但有完整的目录结构，还有一个完整的 Maven 配置，并且生成了一个默认的主程序，几乎所有的准备工作都已经就绪，并且可以立即运行起来（虽然没有提供任何可用的服务）。这也是 Spring Boot 引以为傲的地方，即创建一个应用可以不用编写任何代码，只管运行即可。

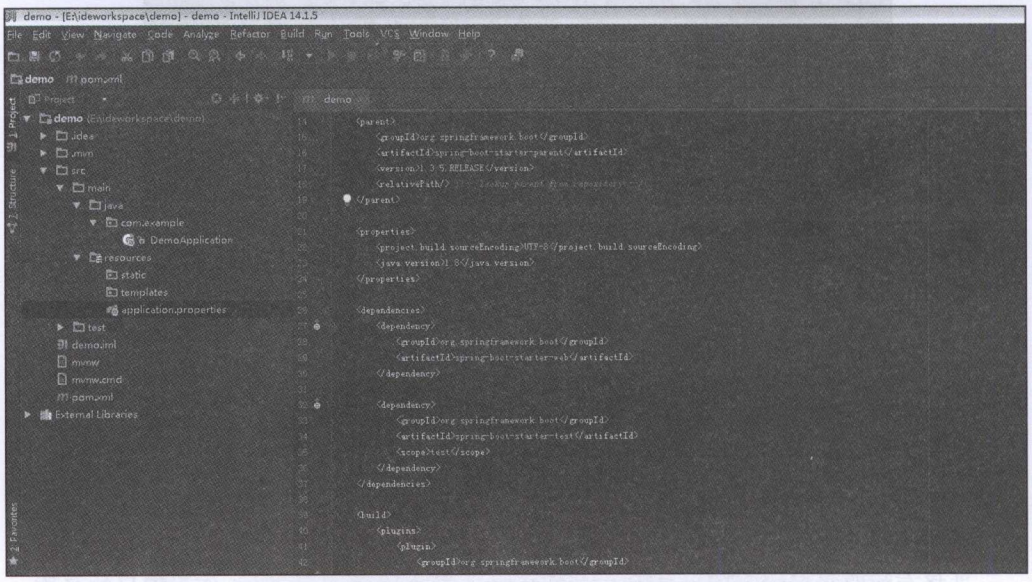


图 1-14 使用 Spring Initializr 创建的初始项目

1.3 使用 Spring Boot

任何应用的开发都需要对项目的创建、运行和发布等进行管理，使用 Spring Boot 框架进行开发，可以选择使用 Maven 或 Gradle 等项目管理工具。在这里我们使用的是 Maven。

1.3.1 Maven 依赖管理

使用 Maven，通过导入 Spring Boot 的 starter 模块，可以将许多程序依赖包自动导入工程中。使用 Maven 的 parent POM，还可以更容易地管理依赖的版本和使用默认的配置，工程中的模块也可以很方便地继承它。例如，使用 1.2.1 节创建的工程，修改 pom.xml 文件，使用如代码清单 1-1 所示的简单 Maven 配置，基本上就能为一个使用 Spring Boot 开发框架的 Web 项目开发提供所需的相关依赖。

代码清单 1-1　Spring Boot Web 基本依赖配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot.example</groupId>
    <artifactId>spring-boot-hello</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.2.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
</project>
```

这里只使用了一个依赖配置 spring-boot-starter-web 和一个 parent 配置 spring-boot-starter-parent，在工程的外部库（External Libraries）列表中，它自动引入的依赖包如代码清单 1-2 所示。

代码清单 1-2　Maven 加载的依赖列表

```
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-starter-web:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-starter:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-autoconfigure:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-starter-logging:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: ch.qos.logback:logback-
classic:1.1.3" level="project" />
    <orderEntry type="library" name="Maven: ch.qos.logback:logback-core:1.1.3"
```

```
level="project" />
    <orderEntry type="library" name="Maven: org.slf4j:slf4j-api:1.7.13"
level="project" />
    <orderEntry type="library" name="Maven: org.slf4j:jcl-over-slf4j:1.7.13"
level="project" />
    <orderEntry type="library" name="Maven: org.slf4j:jul-to-slf4j:1.7.13"
level="project" />
    <orderEntry type="library" name="Maven: org.slf4j:log4j-over-slf4j:1.7.13"
level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
core:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" scope="RUNTIME" name="Maven: org.
yaml:snakeyaml:1.16" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-starter-tomcat:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.apache.tomcat.embed:tomcat-
embed-core:8.0.30" level="project" />
    <orderEntry type="library" name="Maven: org.apache.tomcat.embed:tomcat-
embed-el:8.0.30" level="project" />
    <orderEntry type="library" name="Maven: org.apache.tomcat.embed:tomcat-
embed-logging-juli:8.0.30" level="project" />
    <orderEntry type="library" name="Maven: org.apache.tomcat.embed:tomcat-
embed-websocket:8.0.30" level="project" />
    <orderEntry type="library" name="Maven: org.springframework.boot:spring-
boot-starter-validation:1.3.2.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.hibernate:hibernate-
validator:5.2.2.Final" level="project" />
    <orderEntry type="library" name="Maven: javax.validation:validation-
api:1.1.0.Final" level="project" />
    <orderEntry type="library" name="Maven: org.jboss.logging:jboss-
logging:3.3.0.Final" level="project" />
    <orderEntry type="library" name="Maven: com.fasterxml:classmate:1.1.0"
level="project" />
    <orderEntry type="library" name="Maven: com.fasterxml.jackson.core:jackson-
databind:2.6.5" level="project" />
    <orderEntry type="library" name="Maven: com.fasterxml.jackson.core:jackson-
annotations:2.6.5" level="project" />
    <orderEntry type="library" name="Maven: com.fasterxml.jackson.core:jackson-
core:2.6.5" level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
web:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
aop:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: aopalliance:aopalliance:1.0"
level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
```

```
beans:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
context:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
webmvc:4.2.4.RELEASE" level="project" />
    <orderEntry type="library" name="Maven: org.springframework:spring-
expression:4.2.4.RELEASE" level="project" />
```

在工程的外部库列表中，Spring Boot 已经导入了整个 springframework 依赖，以及 autoconfigure、logging、slf4j、jackson、tomcat 插件等，所有这些都是一个 Web 项目可能需要用到的东西（包括你已经考虑到的和没有考虑到的），它真是一个聪明的助手。

1.3.2　一个简单的实例

Spring Boot 的官方文档中提供了一个最简单的 Web 实例程序，这个实例只使用了几行代码，如代码清单 1-3 所示。虽然简单，但实际上这已经可以算作是一个完整的 Web 项目了。

代码清单 1-3　Spring Boot 简单实例

```
package springboot.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class Application {
    @RequestMapping("/")
    String home() {
        return "hello";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

这个简单实例，首先是一个 Spring Boot 应用的程序入口，或者叫作主程序，其中使用了一个注解 @SpringBootApplication 来标注它是一个 Spring Boot 应用，main 方法

使它成为一个主程序，将在应用启动时首先被执行。其次，注解 @RestController 同时标注这个程序还是一个控制器，如果在浏览器中访问应用的根目录，它将调用 home 方法，并输出字符串：hello。

1.4 运行与发布

本章实例工程的完整代码可以使用 IDEA 直接从 GitHub 的 https://github.com/chenfromsz/spring-boot-hello.git 中检出，如图 1-15 所示，单击 Clone 按钮将整个项目复制到本地。

图 1-15 检出实例工程

1.4.1 在 IDEA 环境中运行

在 IDEA 中打开 Run 菜单，选择 Edit Configuration 打开 Run/Debug Configurations 对话框，在配置界面的左边侧边栏中选择增加一个 Application 或 Spring Boot 配置项目，然后在工作目录中选择工程所在的根目录，主程序选择代码清单 1-3 创建的类：springboot.example.Application，并将配置保存为 hello，如图 1-16 所示。

然后选择 Run 或 Debug 运行 hello 配置项目。如果启动成功，将在控制台中输出类似如下信息：

```
"D:\Program Files\Java\jdk1.8.0_25\bin\java"
.....
 :: Spring Boot ::        (v1.3.2.RELEASE)
......
Starting Servlet Engine: Apache Tomcat/8.0.30
......
Tomcat started on port(s): 8080 (http)
......
```

从上面的输出中可以看出，Tomcat 默认开启了 8080 端口。要访问这个应用提供的

服务，可以在浏览器的地址栏中输入 http://localhost:8080/。这样就可以看到我们期望的输出字符：hello。

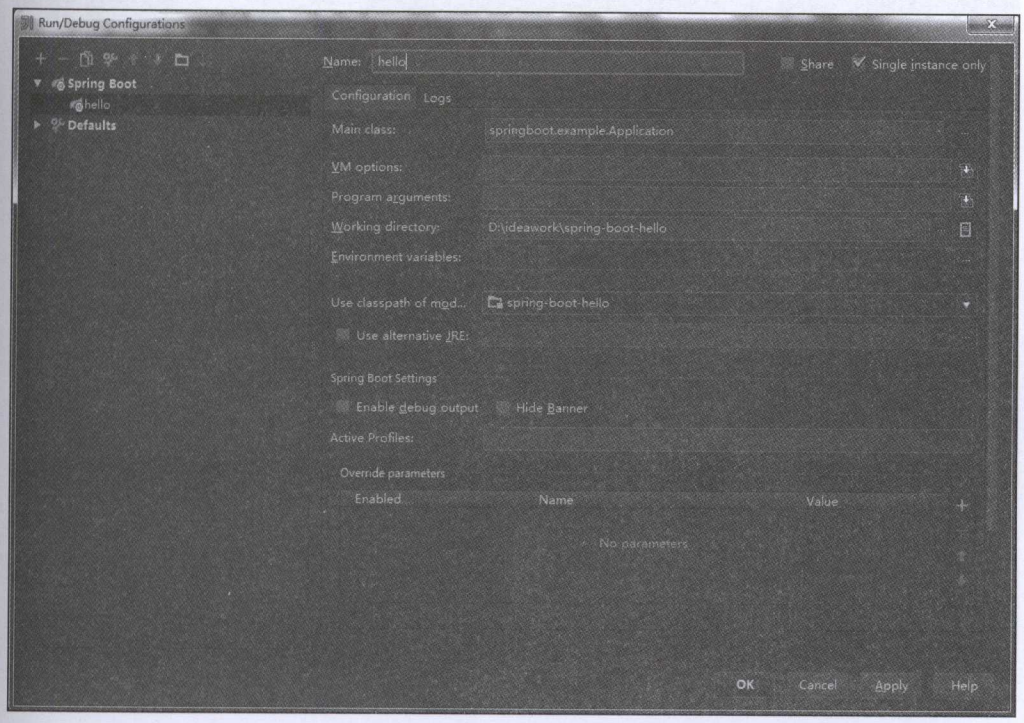


图 1-16　Spring Boot 应用配置

1.4.2　将应用打包发布

上面操作演示了在 IDEA 环境中如何运行一个应用。如果我们想把应用发布出去，需要怎么做呢？可以将代码清单 1-1 中的 Maven 配置增加一个发布插件来实现。如代码清单 1-4 所示，增加了一个打包插件：spring-boot-maven-plugin，并增加了一行打包的配置：<packaging>jar</packaging>，这行配置指定将应用工程打包成 jar 文件。

代码清单 1-4　包含打包插件的 Maven 配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>springboot.example</groupId>
    <artifactId>spring-boot-hello</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.2.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

这样就可以在 IDEA 中增加一个打包的配置，打开 Run/Debug Configurations 对话框，选择增加配置一个 Maven 打包项目，在工作目录中选择工程所在根目录，在命令行中输入 package，并将配置保存为 mvn，如图 1-17 所示。

运行 mvn 打包项目，就可以将实例工程打包，打包的文件将输出在工程的 target 目录中。

如果已经按照 1.1.3 节的说明安装了 Maven，也可以直接使用 Maven 的命令打包。打开一个命令行窗口，将路径切换到工程根目录中，直接在命令行输入 mvn package，

同样也能将项目打包成 jar 文件。执行结果如下：

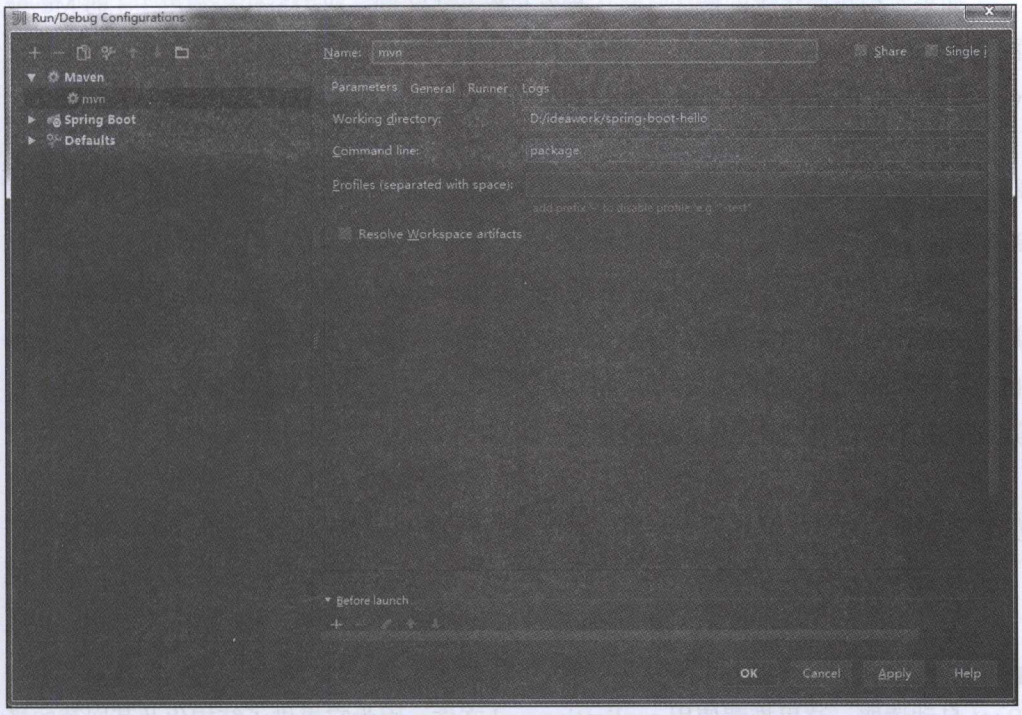


图 1-17　Maven 打包配置

```
......
[INFO] --- maven-jar-plugin:2.5:jar (default-jar) @ spring-boot-hello ---
[INFO] Building jar: E:\ideworkspace\spring-boot-hello\target\spring-boot-hello-
1.0-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.3.2.RELEASE:repackage (default) @ spring-b
oot-hello ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.450 s
[INFO] Finished at: 2016-05-08T16:54:44+08:00
[INFO] Final Memory: 23M/118M
[INFO] ------------------------------------------------------------------------
```

打包成功后，在工程的 target 目录中将会生成 jar 文件 spring-boot-hello-1.0-SNAPSHOT.jar。在命令行窗口中切换到 target 目录中，运行如下指令，就能启动应用。

```
java -jar spring-boot-hello-1.0-SNAPSHOT.jar
```

如果希望按照传统的做法，将工程发布成 war 文件，应当将代码清单 1-4 的 Maven 配置 <packaging>jar</packaging> 改成 <packaging>war</packaging>，这样就可以打包成 war 文件。打包完成后将 war 文件放置在 Tomcat 的 webapp 路径中，启动 Tomcat 就能自动运行程序。

这里需要注意的是，如果自主使用 Tomcat 运行应用，在安装 JDK 时必须配置 JAVA_HOME 环境变量，同时 JDK 要求 1.8 以上的版本，Tomcat 必须是 8.0 以上的版本。

我更加喜欢打包成 jar，然后使用 Spring Boot 的嵌入插件 Tomcat 运行应用。本书所有实例都可以打包成 jar 直接运行。即使对于一个包含很多页面、图片、脚本等资源的复杂应用系统，这种方法也是可行的，并且打包成 jar，更方便项目发布在 Docker 上运行，这些将在后面的章节中详细介绍。

1.5 关于 Spring Boot 配置

关于 Spring Boot 配置，可以在工程的 resources 文件夹中创建一个 application.properties 或 application.yml 文件，这个文件会被发布在 classpath 中，并且被 Spring Boot 自动读取。这里推荐使用 application.yml 文件，因为它提供了结构化及其嵌套的格式，例如，可以按如下所示配置上面的工程，将默认端口改为 80，并且将 Tomcat 的字符集定义为 UTF-8。

```
server:
    port: 80
    tomcat:
        uri-encoding: UTF-8
```

如果要使用 application.properties 文件，上面的配置就要改成如下所示的样子，其结果完全相同。

```
server.port = 80
server.tomcat.uri-enconding = UTF-8
```

使用这个配置文件可以直接使用 Spring Boot 预定义的一些配置参数，关于其他配置参数的详细说明和描述可以查看官方的文档说明：https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html。在后面的开发中将在用

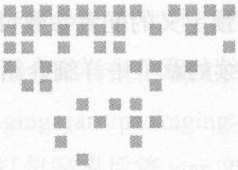
得到的地方选择使用这些预定义的配置参数。即使没有预定义的配置参数可用，也能很容易地按照应用的需要自定义一些配置参数，这将在后续的章节中详细介绍。

1.6　小结

本章主要介绍了 Spring Boot 开发环境的搭建，以及一些开发工具的安装配置，内容难免有点枯燥。然后创建并运行一个非常简单的实例工程，让性急的读者一睹 Spring Boot 的芳容。

本章实例工程只是使用 Spring Boot 框架进行开发的非常简单的入门指引。因为 Spring Boot 开发框架是一个非常轻量级的开发框架，所以也有人把它叫作微框架，从入门指引中可以看出，使用 Spring Boot 框架开发应用不但入门容易，而且其蕴藏的无比强大的功能，使开发过程也变得更加容易。

下面，让我们使用 Spring Boot 框架进行一些更加有趣的开发吧。这一章只是小试牛刀而已，在后续章节中将使用 Spring Boot 框架来开始一些真正的开发。



Chapter 2 第 2 章

在 Spring Boot 中使用数据库

使用数据库是开发基本应用的基础。借助于开发框架，我们已经不用编写原始的访问数据库的代码，也不用调用 JDBC（Java Data Base Connectivity）或者连接池等诸如此类的被称作底层的代码，我们将在高级的层次上访问数据库。而 Spring Boot 更是突破了以前所有开发框架访问数据库的方法，在前所未有的更加高级的层次上访问数据库。因为 Spring Boot 包含一个功能强大的资源库，为使用 Spring Boot 的开发者提供了更加简便的接口进行访问。

本章将介绍怎样使用传统的关系型数据库，以及近期一段时间异军突起的 NoSQL（Not Only SQL）数据库。

本章的实例工程使用了分模块的方式构建，各模块的定义如表 2-1 所示。

表 2-1 实例工程模块定义

项　目	工　程	功　能
MySQL 模块	mysql	使用 MySQL
Redis 模块	redis	使用 Redis
MongoDB 模块	mongodb	使用 MongoDB
Neo4j 模块	neo4j	使用 Neo4j

2.1 使用 MySQL

对于传统关系型数据库来说，Spring Boot 使用 JPA（Java Persistence API）资源库

来实现对数据库的操作，使用 MySQL 也是如此。简单地说，JPA 就是为 POJO（Plain Ordinary Java Object）提供持久化的标准规范，即将 Java 的普通对象通过对象关系映射（Object-Relational Mapping，ORM）持久化到数据库中。

2.1.1　MySQL 依赖配置

为了使用 JPA 和 MySQL，首先在工程中引入它们的 Maven 依赖，如代码清单 2-1 所示。其中，指定了在运行时调用 MySQL 的依赖。

代码清单 2-1　JPA 和 Mysql 依赖配置

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

2.1.2　实体建模

首先创建一些普通对象，用来与数据库的表建立映射关系，接着演示如何使用 JPA 对数据库进行增删查改等存取操作。

假如现在有三个实体：部门、用户和角色，并且它们具有一定的关系，即一个用户只能隶属于一个部门，一个用户可以拥有多个角色。它们的关系模型如图 2-1 所示。

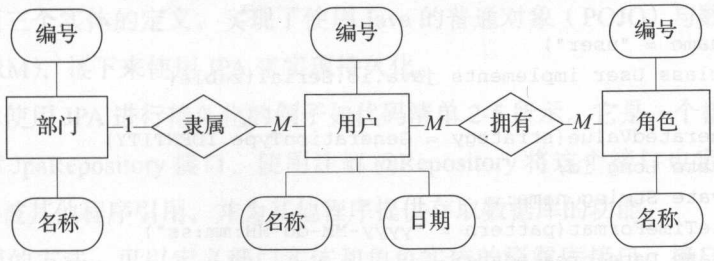


图 2-1　MySQL 实体 – 关系模型示例

Spring Boot 的实体建模与使用 Spring 框架时的定义方法一样，同样比较方便的是

使用了注解的方式来实现。

部门实体的建模如代码清单 2-2 所示，其中注解 @Table 指定关联的数据库的表名，注解 @Id 定义一条记录的唯一标识，并结合注解 @GeneratedValue 将其设置为自动生成。部门实体只有两个字段：id 和 name。程序中省略了 Getter 和 Setter 方法的定义，这些方法可以使用 IDEA 的自动生成工具很方便地生成。

代码清单 2-2 部门实体建模

```
@Entity
@Table(name = "deparment")
public class Deparment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    public Deparment() {
    }
    ......
}
```

用户实体包含三个字段：id、name 和 createdate，用户实体建模如代码清单 2-3 所示。其中注解 @ManyToOne 定义它与部门的多对一关系，并且在数据库表中用字段 did 来表示部门的 ID，注解 @ManyToMany 定义与角色实体的多对多关系，并且用中间表 user_role 来存储它们各自的 ID，以表示它们的对应关系。日期类型的数据必须使用注解 @DateTimeFormat 来进行格式化，以保证它在存取时能提供正确的格式，避免保存失败。注解 @JsonBackReference 用来防止关系对象的递归访问。

代码清单 2-3 用户实体建模

```
@Entity
@Table(name = "user")
public class User implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date createdate;

    @ManyToOne
```

```
    @JoinColumn(name = "did")
    @JsonBackReference
    private Department deparment;

    @ManyToMany(cascade = {}, fetch = FetchType.EAGER)
    @JoinTable(name = "user_role",
            joinColumns = {@JoinColumn(name = "user_id")},
            inverseJoinColumns = {@JoinColumn(name = "roles_id")})
    private List<Role> roles;

    public User() {
    }
.....
```

角色实体建模比较简单，只要按设计的要求，定义 id 和 name 字段即可，当然同样必须保证 id 的唯一性并将其设定为自动生成。角色实体的建模如代码清单 2-4 所示。

代码清单 2-4　角色实体建模

```
@Entity
@Table(name = "role")
public class Role implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    public Role() {
    }
.....
```

2.1.3　实体持久化

通过上面三个实体的定义，实现了使用 Java 的普通对象（POJO）与数据库表建立映射关系（ORM），接下来使用 JPA 来实现持久化。

用户实体使用 JPA 进行持久化的例子如代码清单 2-5 所示。它是一个接口，并继承于 JPA 资源库 JpaRepository 接口，使用注解 @Repository 将这个接口也定义为一个资源库，使它能被其他程序引用，并为其他程序提供存取数据库的功能。

使用相同的方法，可以定义部门实体和角色实体的资源库接口。接口同样继承于 JpaRepository 接口，只要注意使用的参数是各自的实体对象即可。

代码清单 2-5 用户实体持久化

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

这样就实现存取数据库的功能了。现在可以对数据库进行增删查改、进行分页查询和指定排序的字段等操作。

或许你还有疑问，我们定义的实体资源库接口并没有声明一个方法，也没有对接口有任何实现的代码，甚至连一条 SQL 查询语句都没有写，这怎么可能？

是的，使用 JPA 就是可以这么简单。我们来看看 JpaRepository 的继承关系，你也许会明白一些。如图 2-2 所示，JpaRepository 继承于 PagingAndSortingRepository，它提供了分页和排序功能，PagingAndSortingRepository 继承于 CrudRepository，它提供了简单的增删查改功能。

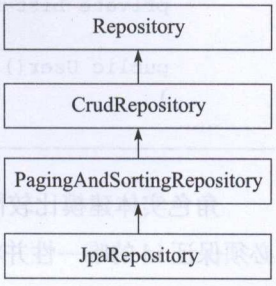


图 2-2 JpaRepository 接口继承关系

因为定义的接口继承于 JpaRepository，所以它传递性地继承上面所有这些接口，并拥有这些接口的所有方法，这样就不难理解为何它包含那么多功能了。这些接口提供的一些方法如下：

```
<S extends T> S save(S var1);
T findOne(ID var1);
long count();
void delete(ID var1);
void delete(T var1);
void deleteAll();
Page<T> findAll(Pageable var1);
List<T> findAll();
List<T> findAll(Sort var1);
List<T> findAll(Iterable<ID> var1);
void deleteAllInBatch();
T getOne(ID var1);
......
```

JPA 还提供了一些自定义声明方法的规则，例如，在接口中使用关键字 findBy、readBy、getBy 作为方法名的前缀，拼接实体类中的属性字段（首个字母大写），并可选择拼接一些 SQL 查询关键字来组合成一个查询方法。例如，对于用户实体，下列查询关键字可以这样使用：

- And，例如 findByIdAndName（Long id，String name）；
- Or，例如 findByIdOrName（Long id，String name）；
- Between，例如 findByCreatedateBetween（Date start，Date end）；
- LessThan，例如 findByCreatedateLessThan（Date start）；
- GreaterThan，例如 findByCreatedateGreaterThan（Date start）；
- IsNull，例如 findByNameIsNull()；
- IsNotNull，例如 findByNameIsNotNull()；
- NotNull，与 IsNotNull 等价；
- Like，例如 findByNameLike（String name）；
- NotLike，例如 findByNameNotLike（String name）；
- OrderBy，例如 findByNameOrderByIdAsc（String name）；
- Not，例如 findByNameNot（String name）；
- In，例如 findByNameIn（Collection<String>nameList）；
- NotIn，例如 findByNameNotIn（Collection<String>nameList）。

又如下列对用户实体类自定义的方法声明，它们都是符合 JPA 规则的，这些方法也不用实现，JPA 将会代理实现这些方法。

```
User findByNameLike(String name);
User readByName(String name);
List<User> getByCreatedateLessThan(Date star);
```

2.1.4　MySQL 测试

现在，为了验证上面设计的正确性，我们用一个实例来测试一下。

首先，增加一个使用 JPA 的配置类，如代码清单 2-6 所示。其中 @EnableTransactionManagement 启用了 JPA 的事务管理；@EnableJpaRepositories 启用了 JPA 资源库并指定了上面定义的接口资源库的位置；@EntityScan 指定了定义实体的位置，它将导入我们定义的实体。注意，在测试时使用的 JPA 配置类可能与这个配置略有不同，这个配置的一些配置参数是从配置文件中读取的，而测试时使用的配置类把一些配置参数都包含在类定义中了。

代码清单 2-6　JPA 配置类

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@EnableTransactionManagement(proxyTargetClass = true)
```

```
@EnableJpaRepositories(basePackages = "dbdemo.**.repository")
@EntityScan(basePackages = "dbdemo.**.entity")
public class JpaConfiguration {

    @Bean
     PersistenceExceptionTranslationPostProcessor persistenceExceptionTrans
lationPostProcessor(){
        return new PersistenceExceptionTranslationPostProcessor();
    }

}
```

其次，在 MySQL 数据库服务器中创建一个数据库 test，然后配置一个可以访问这个数据库的用户及其密码。数据库的表结构可以不用创建，在程序运行时将会按照实体的定义自动创建。如果还没有创建一个具有完全权限访问数据库 test 的用户，可以在连接 MySQL 服务器的查询窗口中执行下面指令，这个指令假设你将在本地中访问数据库。

```
grant all privileges on test.* to 'root'@'localhost' identified by
'12345678';
```

然后，在 Spring Boot 的配置文件 application.yml 中使用如代码清单 2-7 所示的配置，用来设置数据源和 JPA 的工作模式。

代码清单 2-7 数据源和 JPA 配置

```
spring:
    datasource:
        url: jdbc:mysql://localhost:3306/test?characterEncoding=utf8
        username: root
        password: 12345678
    jpa:
        database: MYSQL
        show-sql: true
    #Hibernate ddl auto (validate|create|create-drop|update)
        hibernate:
            ddl-auto: update
            naming-strategy: org.hibernate.cfg.ImprovedNamingStrategy
        properties:
            hibernate:
                dialect: org.hibernate.dialect.MySQL5Dialect
```

配置中将 ddl-atuo 设置为 update，就是使用 Hibernate 来自动更新表结构的，即如

果数据表不存在则创建，或者如果修改了表结构，在程序启动时则执行表结构的同步更新。

最后，编写一个测试程序，如代码清单 2-8 所示。测试程序首先初始化数据库，创建一个部门，命名为“开发部”，创建一个角色，命名为 admin，创建一个用户，命名为 user，同时将它的所属部门设定为上面创建的部门，并将现有的所有角色都分配给这个用户。然后使用分页的方式查询所有用户的列表，并从查到的用户列表中，打印出用户的名称、部门的名称和第一个角色的名称等信息。

代码清单 2-8　MySQL 测试程序

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JpaConfiguration.class})
public class MysqlTest {
    private static Logger logger = LoggerFactory.getLogger(MysqlTest.class);

    @Autowired
    UserRepository userRepository;
    @Autowired
    DepartmentRepository departmentRepository;
    @Autowired
    RoleRepository roleRepository;

    @Before
    public void initData(){
        userRepository.deleteAll();
        roleRepository.deleteAll();
        departmentRepository.deleteAll();

        Department department = new Department();
        department.setName("开发部");
        departmentRepository.save(department);
        Assert.notNull(department.getId());

        Role role = new Role();
        role.setName("admin");
        roleRepository.save(role);
        Assert.notNull(role.getId());

        User user = new User();
        user.setName("user");
        user.setCreatedate(new Date());
        user.setDeparment(department);
```

```
            List<Role> roles = roleRepository.findAll();
            Assert.notNull(roles);
            user.setRoles(roles);

            userRepository.save(user);
            Assert.notNull(user.getId());
        }

        @Test
        public void findPage(){
            Pageable pageable = new PageRequest(0, 10, new Sort(Sort.Direction.ASC,
"id"));
            Page<User> page = userRepository.findAll(pageable);
            Assert.notNull(page);
            for(User user : page.getContent()) {
                logger.info("====user==== user name:{}, department name:{}, role
name:{}",
                        user.getName(), user.getDeparment().getName(), user.getRoles().
get(0).getName());
            }
        }
    }
```

好了，现在可以使用JUnit来运行这个测试程序了，在IDEA的Run/Debug Configuration配置中增加一个JUint配置项，模块选择mysql，工作目录选择模块所在的根目录，程序选择dbdemo.mysql.test.MysqlTest，并将配置项目名称保存为mysqltest，如图2-3所示。

用Debug方式运行测试配置项目mysqltest，可以在控制台中看到执行的过程和结果。如果状态栏中显示为绿色，并且提示“All Tests passed”，则表示测试全部通过。在控制台中也可以查到下列打印信息：

```
dbdemo.mysql.test.MysqlTest - ====user==== user name:user, department
name:开发部, role name:admin
```

这时如果在MySQL服务器中查看数据库test，不但可以看到表结构都已经创建了，还可以看到上面测试生成的一些数据。

这是不是很激动人心？在Spring Boot使用数据库，就是可以如此简单和有趣。到目前为止，我们不仅没有写过一条查询语句，也没有实现一个访问数据库的方法，但是已经能对数据库执行所有的操作，包括一般的增删查改和分页查询。

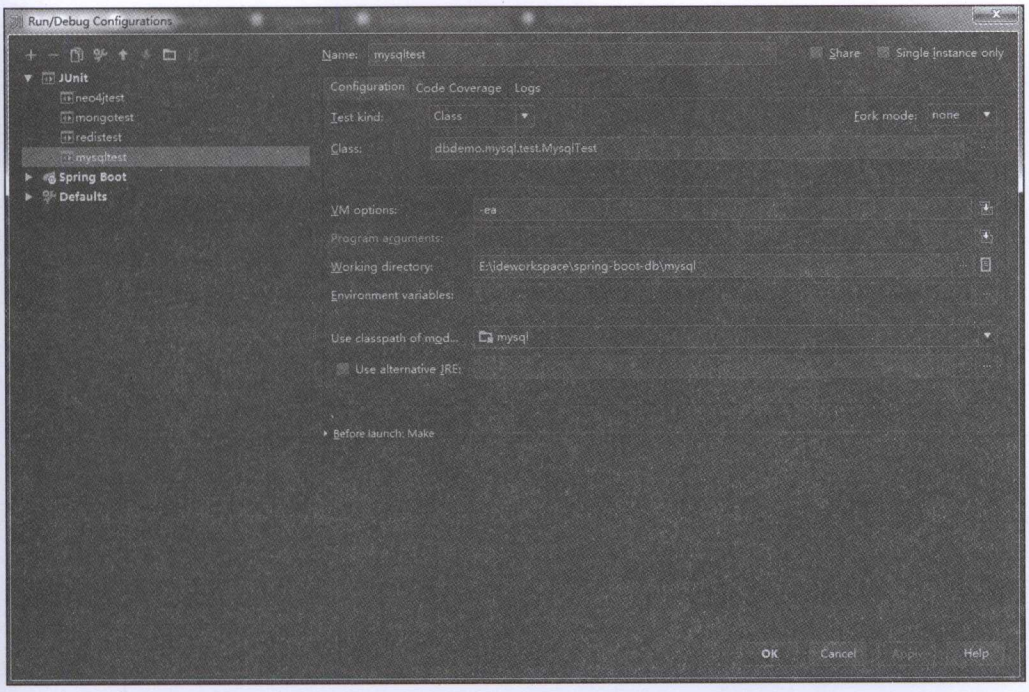


图 2-3　JUint 测试配置

2.2　使用 Redis

关系型数据库在性能上总是存在一些这样那样的缺陷，所以大家有时候在使用传统关系型数据库时，会与具有高效存取功能的缓存系统结合使用，以提高系统的访问性能。在很多流行的缓存系统中，Redis 是一个不错的选择。Redis 是一种可以持久存储的缓存系统，是一个高性能的 key-value 数据库，它使用键 – 值对的方式来存储数据。

2.2.1　Redis 依赖配置

需要使用 Redis，可在工程的 Maven 配置中加入 spring-boot-starter-redis 依赖，如代码清单 2-9 所示。其中 gson 是用来转换 Json 数据格式的工具，mysql 是引用了上一节的模块，这里使用 2.1 节定义的实体对象来存取数据，演示在 Redis 中的存取操作。

代码清单 2-9　Redis 模块的 Maven 依赖配置

```
<dependencies>
    <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-redis</artifactId>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.2.4</version>
        </dependency>
        <dependency>
            <groupId>springboot.db</groupId>
            <artifactId>mysql</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
```

2.2.2 创建 Redis 服务类

Redis 提供了下列几种数据类型可供存取：

- string；
- hash；
- list；
- set 及 zset。

在实例中，将使用 string 即字符串的类型来演示数据的存取操作。对于 Redis，Spring Boot 没有提供像 JPA 那样相应的资源库接口，所以只能仿照上一节中 Repository 的定义编写一个实体 User 的服务类，如代码清单 2-10 所示。这个服务类可以存取对象 User 以及由 User 组成的列表 List，同时还提供了一个删除的方法。所有这些方法都是使用 RedisTemplate 来实现的。

代码清单 2-10 用户实体的 Redis 服务类

```
    @Repository
    public class UserRedis {
        @Autowired
        private RedisTemplate<String, String> redisTemplate;

        public void add(String key, Long time,User user) {
            Gson gson = new Gson();
            redisTemplate.opsForValue().set(key, gson.toJson(user), time, TimeUnit.
MINUTES);
        }
```

```
    public void add(String key, Long time, List<User> users) {
        Gson gson = new Gson();
        redisTemplate.opsForValue().set(key, gson.toJson(users), time, TimeUnit.
MINUTES);
    }

    public User get(String key) {
        Gson gson = new Gson();
        User user = null;
        String userJson = redisTemplate.opsForValue().get(key);
        if(!StringUtils.isEmpty(userJson))
            user = gson.fromJson(userJson, User.class);
        return user;
    }

    public List<User> getList(String key) {
        Gson gson = new Gson();
        List<User> ts = null;
        String listJson = redisTemplate.opsForValue().get(key);
        if(!StringUtils.isEmpty(listJson))
            ts = gson.fromJson(listJson, new TypeToken<List<User>>(){}.getType());
        return ts;
    }

    public void delete(String key){
        redisTemplate.opsForValue().getOperations().delete(key);
    }
}
```

Redis 没有表结构的概念，所以要实现 MySQL 数据库中表的数据（即普通 Java 对象映射的实体数据）在 Redis 中存取，必须做一些转换，使用 JSON 格式的文本作为 Redis 与 Java 普通对象互相交换数据的存储格式。这里使用 Gson 工具将类对象转换为 JSON 格式的文本进行存储，要取出数据时，再将 JSON 文本数据转化为 Java 对象。

因为 Redis 使用了 key-value 的方式存储数据，所以存入时要生成一个唯一的 key，而要查询或者删除数据时，就可以使用这个唯一的 key 进行相应的操作。

保存在 Redis 数据库中的数据默认是永久存储的，可以指定一个时限来确定数据的生命周期，超过指定时限的数据将被 Redis 自动清除。在代码清单 2-10 中我们以分钟为单位设定了数据的存储期限。

另外，为了能正确调用 RedisTemplate，必须对其进行一些初始化工作，即主要对它存取的字符串进行一个 JSON 格式的系列化初始配置，如代码清单 2-11 所示。

代码清单 2-11 RedisTemplate 初始化

```
@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String, String> redisTemplate(
            RedisConnectionFactory factory) {
        StringRedisTemplate template = new StringRedisTemplate(factory);
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jack
son2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        template.setValueSerializer(jackson2JsonRedisSerializer);
        template.afterPropertiesSet();
        return template;
    }

}
```

2.2.3 Redis 测试

如果还没有安装 Redis 服务器，可以参照本书附录 C 提供的方法安装，然后在工程的配置文件 application.yml 中配置连接 Redis 服务器等参数，如代码清单 2-12 所示。其中 host 和 port 分别表示 Redis 数据库服务器的 IP 地址和开放端口，database 可以不用指定，由 Redis 根据存储情况自动选定（注：测试时这些配置是集成在一个配置类中实现的）。

代码清单 2-12 Redis 配置

```
spring:
    redis:
    # database: 1
        host: 192.168.1.214
        port: 6379
        pool:
            max-idle: 8
            min-idle: 0
            max-active: 8
            max-wait: -1
```

现在编写一个 JUint 测试程序，来演示如何在 Redis 服务器中存取数据，如代码清

单 2-13 所示。测试程序创建一个部门对象并将其命名为“开发部”，创建一个角色对象并把它命名为 admin，创建一个用户对象并把它命名为 user，同时设定这个用户属于“开发部”，并把 admin 这个角色分配给这个用户。接着测试程序使用类名等参数生成一个 key，并使用这个 key 清空原来的数据，然后用这个 key 存储现在这个用户的数据，最后使用这个 key 查询用户，并将查到的信息打印出来。

代码清单 2-13　Redis 测试程序

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {RedisConfig.class, UserRedis.class})
public class RedisTest {
    private static Logger logger = LoggerFactory.getLogger(RedisTest.class);

    @Autowired
    UserRedis userRedis;

    @Before
    public void setup(){
        Deparment deparment = new Deparment();
        deparment.setName("开发部");

        Role role = new Role();
        role.setName("admin");

        User user = new User();
        user.setName("user");
        user.setCreatedate(new Date());
        user.setDeparment(deparment);

        List<Role> roles = new ArrayList<>();
        roles.add(role);

        user.setRoles(roles);

        userRedis.delete(this.getClass().getName()+":userByname:"+user.getName());
        userRedis.add(this.getClass().getName()+":userByname:"+user.getName(),
10L, user);

    }

    @Test
    public void get(){
        User user = userRedis.get(this.getClass().getName()+":userByname:
user");
```

```
            Assert.notNull(user);
            logger.info("======user====== name:{}, deparment:{}, role:{}",
                user.getName(), user.getDeparment().getName(), user.getRoles().get(0).
getName());
        }
    }
```

要运行这个测试程序，可以在 IDEA 的 Run/Debug Configuration 配置中增加一个 JUint 配置项目，模块选择 redis，工作目录选择模块所在的根目录，类选择这个测试程序即 dbdemo.redis.test.RedisTest，并将配置保存为 redistest。

使用 Debug 方式运行测试项目 redistest。如果测试通过，会输出一个用户的用户名、所属部门和拥有角色等简要信息，如下所示：

```
    dbdemo.redis.test.RedisTest - ======user====== name:user, deparment:开发部,
role:admin
```

对于 Redis 的使用，还可以将注解方式与调用数据库的方法相结合，那样就不用再编写像上面那样的服务类，并且使用起来更加简单，这将在后面的章节中介绍。

2.3 使用 MongoDB

在当前流行的 NoSQL 数据库中，MongoDB 是大家接触比较早而且用得比较多的数据库。MongoDB 是文档型的 NoSQL 数据库，具有大数据量、高并发等优势，但缺点是不能建立实体关系，而且也没有事务管理机制。

2.3.1 MongoDB 依赖配置

在 Spring Boot 中使用 MongoDB 也像使用 JPA 一样容易，并且同样拥有功能完善的资源库。同样的，要使用 MongoDB，首先必须在工程的 Maven 中引入它的依赖，如代码清单 2-14 所示。除了 MongoDB 本身的依赖之外，还需要一些附加的工具配套使用。

代码清单 2-14 使用 MongoDB 的 Maven 依赖配置

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-mongodb</artifactId>
```

```
    </dependency>
    <dependency>
        <groupId>org.pegdown</groupId>
        <artifactId>pegdown</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-hateoas</artifactId>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
    </dependency>
</dependencies>
```

2.3.2　文档建模

MongoDB 是文档型数据库，使用 MongoDB 也可以像使用关系型数据库那样为文档建模。如代码清单 2-15 所示，为用户文档建模，它具有用户名、密码、用户名称、邮箱和注册日期等字段，有一个用来保存用户角色的数据集，还定义了一个构造函数，可以很方便地用来创建一个用户实例。

代码清单 2-15　用户文档建模

```
@Document(collection = "user")
public class User {
    @Id
    private String userId;
    @NotNull @Indexed(unique = true)
    private String username;
    @NotNull
    private String password;
    @NotNull
    private String name;
    @NotNull
    private String email;
    @NotNull
    private Date registrationDate = new Date();
    private Set<String> roles = new HashSet<>();

    public User() { }

    @PersistenceConstructor
```

```
        public User(String userId, String username, String password, String name,
String email,
                Date registrationDate, Set<String> roles) {
            this.userId = userId;
            this.username = username;
            this.password = password;
            this.name = name;
            this.email = email;
            this.registrationDate = registrationDate;
            this.roles = roles;

        }
    ......
```

2.3.3 文档持久化

MongoDB 也有像使用 JPA 那样的资源库，如代码清单 2-16 所示，为用户文档创建了一个 Repository 接口，继承于 MongoRepository，实现了文档持久化。

代码清单 2-16 用户文档持久化

```
public interface UserRepository extends MongoRepository<User, String> {
    User findByUsername(String username);
}
```

MongoRepository 的继承关系如图 2-4 所示，看起来跟 JPA 的资源库的继承关系没有什么两样，它也包含访问数据库的丰富功能。

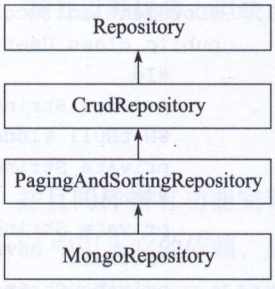


图 2-4 MongoRepository 接口继承关系

代码清单 2-17 是用在测试中的使用 MongoDB 的一个配置类定义，其中 @PropertySource 指定读取数据库配置文件的位置和名称，@EnableMongoRepositories 启用资源库并设定定义资源库接口放置的位置，这里使用环境变量 Environment 来读取配置文件的一些数据库配置参数，然后使用一个数据库客户端，连接 MongoDB 服务器。

代码清单 2-17 TestDataSourceConfig 配置类

```
@Configuration
@EnableMongoRepositories(basePackages = "dbdemo.mongo.repositories")
@PropertySource("classpath:test.properties")
```

```
public class TestDataSourceConfig extends AbstractMongoConfiguration {

    @Autowired private Environment env;

    @Override
    public String getDatabaseName(){
        return env.getRequiredProperty("mongo.name");
    }

    @Override
    @Bean
    public Mongo mongo() throws Exception {
        ServerAddress serverAddress = new ServerAddress(env.getRequiredProperty
("mongo.host"));
        List<MongoCredential> credentials = new ArrayList<>();
        return new MongoClient(serverAddress, credentials);
    }

}
```

2.3.4　MongoDB 测试

如果还没有安装 MongoDB 服务器，可以参照附录 B 的方法安装并启动一个 MongoDB 服务器。然后，使用如代码清单 2-18 所示的配置方法配置连接服务器的一些参数，该配置假定你的 MongoDB 服务器安装在本地，并使用默认的数据库端口：27017。

代码清单 2-18　MongoDB 数据库配置

```
# MongoDB
mongo.host=localhost
mongo.name=test
mongo.port=27017
```

这样就可以编写一个 JUint 测试例子来测试 UserRepository 接口的使用情况，如代码清单 2-19 所示。测试例子首先使用用户文档类创建一个用户对象实例，然后使用资源库接口调用 save 方法将用户对象保存到数据库中，最后使用 findAll 方法查询所有用户的列表，并使用一个循环输出用户的简要信息。

代码清单 2-19　MongoDB 测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataSourceConfig.class})
@FixMethodOrder
```

```
    public class RepositoryTests {
        private static Logger logger = LoggerFactory.getLogger(RepositoryTests.class);

        @SuppressWarnings("SpringJavaAutowiringInspection") @Autowired
        UserRepository userRepository;

        @Before
        public void setup(){
            Set<String> roles = new HashSet<>();
            roles.add("manage");
            User user = new User("1","user","12345678","name","email@com.cn",new Date(),
roles);
            userRepository.save(user);
        }

        @Test
        public void findAll(){
            List<User> users = userRepository.findAll();
            Assert.notNull(users);
            for(User user : users){
                logger.info("===user=== userid:{}, username:{}, pass:{}, registra
tionDate:{}",
                    user.getUserId(), user.getName(), user.getPassword(), user.
getRegistrationDate());
            }
        }
    }
```

现在可以在 IDEA 的 Run/Debug Configuration 配置中增加一个 JUint 测试项目，模块选择 mongodb，工作目录选择模块所在的工程根目录，类选择上面编写的测试例子，即 dbdemo.mongo.test.RepositoryTests，并将配置保存为 mongotest。

使用 Debug 方式运行测试项目 mongotest。如果通过测试，将输出查到的用户的简要信息，如下所示：

```
    dbdemo.mongo.test.RepositoryTests - ===user=== userid:1, username:name,
pass:12345678, registrationDate:Tue Jun 07 14:26:02 CST 2016
```

这时使用 MongoDB 数据库客户端输入下面的查询指令，也可以查到这条文档的详细信息，这是一条 JSON 结构的文本信息。

```
    > db.user.find()
    { "_id" : "1", "_class" : "dbdemo.mongo.models.User", "username" :
"user", "password" : "12345678", "name" : "name", "email" : "email@com.cn",
```

```
"registrationDate" : ISODate("2016-04-13T06:27:02.423Z"), "roles" : [ "manage" ] }
```

2.4　使用 Neo4j

有没有既具有传统关系型数据库的优点，又具备 NoSQL 数据库优势的一种数据库呢？Neo4j 就是一种这样的数据库。Neo4j 是一个高性能的 NoSQL 图数据库，并且具备完全事务特性。Neo4j 将结构化数据存储在一张图上，图中每一个节点的属性表示数据的内容，每一条有向边表示数据的关系。Neo4j 没有表结构的概念，它的数据用节点的属性来表示。

2.4.1　Neo4j 依赖配置

在 Spring Boot 中使用 Neo4j 非常容易，因为有 spring-data-neo4j 提供了强大的支持。首先，在工程的 Maven 管理中引入 Neo4j 的相关依赖，如代码清单 2-20 所示。

代码清单 2-20　使用 Neo4j 的 Maven 依赖配置

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-rest</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.voodoodyne.jackson.jsog</groupId>
        <artifactId>jackson-jsog</artifactId>
        <version>1.1</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

2.4.2　节点和关系实体建模

虽然 Neo4j 没有表结构的概念，但它有节点和关系的概念。例如，现在有演员和电影两个实体，它们的关系表现为一个演员在一部电影中扮演一个角色。那么就可以创建演员和电影两个节点实体，和一个角色关系实体。它们的实体 – 关系模型如图 2-5 所

示。这个实体－关系模型的定义比起关系型数据库的实体－关系模型的定义要简单得多，但是它更加形象和贴切地表现了实体之间的关系。更难能可贵的是，这个实体－关系模型是可以不经过任何转换而直接存入数据库的，也就是说，在 Neo4j 图数据库中保存的数据与图 2-5 所示的相同，它仍然是一张图。这对于业务人员和数据库设计人员来说，它的意义相同。所以使用 Neo4j 数据库，将在很大程度上减轻了设计工作和沟通成本。

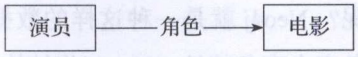


图 2-5 Neo4j 实体－关系模型示例

像 JPA 使用了 ORM 一样，Neo4j 使用了对象－图形映射（Object-Graph Mapping，OGM）的方式来建模。代码清单 2-21 是演员节点实体建模，使用注解 @JsonIdentityInfo 是防止查询数据时引发递归访问效应，注解 @NodeEntity 标志这个类是一个节点实体，注解 @GraphId 定义了节点的一个唯一性标识，它将在创建节点时由系统自动生成，所以它是不可缺少的。这个节点预定义了其他两个属性，name 和 born。节点的属性可以随需要增加或减少，这并不影响节点的使用。

代码清单 2-21 演员节点实体建模

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@NodeEntity
public class Actor {
    @GraphId Long id;
    private String name;
    private int born;

    public Actor() { }
……
```

代码清单 2-22 是电影节点实体建模，注解 @Relationship 表示 List<Role> 是一个关系列表，其中 type 设定了关系的类型，direction 设定这个关系的方向，Relationship.INCOMING 表示以这个节点为终点。addRole 定义了增加一个关系的方法。

代码清单 2-22 电影节点实体建模

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@NodeEntity
public class Movie {
    @GraphId Long id;
    String title;
    String year;
    String tagline;
    @Relationship(type="ACTS_IN", direction = Relationship.INCOMING)
```

```
    List<Role> roles = new ArrayList<>();

    public Role addRole(Actor actor, String name){
        Role role = new Role(actor,this,name);
        this.roles.add(role);
        return role;
    }

    public Movie() { }
.....
```

代码清单 2-23 是角色的关系实体建模，注解 @RelationshipEntity 表明这个类是一个关系实体，并用 type 指定了关系的类型，其中 @StartNode 指定起始节点的实体，@EndNode 指定终止节点的实体，这说明了图中一条有向边的起点和终点的定义。其中定义了一个创建关系的构造函数 Role（Actor actor，Movie movie，String name），这里的 name 参数用来指定这个关系的属性。

代码清单 2-23　角色关系实体建模

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@RelationshipEntity(type = "ACTS_IN")
public class Role {
    @GraphId
    Long id;
    String role;
    @StartNode
    Actor actor;
    @EndNode
    Movie movie;

    public Role() {
    }

    public Role(Actor actor, Movie movie, String name) {
        this.actor = actor;
        this.movie = movie;
        this.role = name;
    }
.....
```

2.4.3　节点实体持久化

像对其他数据库的访问和存取等操作一样，spring-data-neo4j 提供了功能丰富的资

源库可供调用，因此，对于演员和电影节点实体，可以创建它们对应的资源库接口，实现实体的持久化。代码清单 2-24 是电影资源库接口的定义，它继承于 GraphRepository 接口，实现了电影实体的持久化。使用相同方法可以对演员的节点实体实现持久化。关系实体却不用实现持久化，当保存节点实体时，节点实体的关系将会同时保存。

代码清单 2-24 电影实体持久化

```
@Repository
public interface MovieRepository extends GraphRepository<Movie> {
    Movie findByTitle(@Param("title") String title);
}
```

其中 GraphRepository 接口的继承关系也遵循了 Spring Boot 资源库定义的规则，即使用与 JPA 相同的标准规范，所以它同样包含使用数据库的丰富功能，如图 2-6 所示。

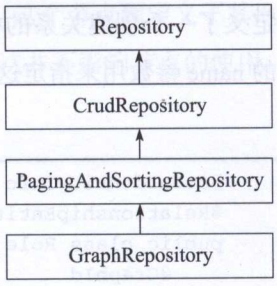


图 2-6 GraphRepository 接口继承关系

2.4.4 Neo4j 测试

代码清单 2-24 是 Neo4j 的数据库配置类，其中 @EnableTransactionManagement 启用了事务管理，@EnableNeo4jRepositories 启用了 Neo4j 资源库并指定了我们定义的资源库接口的位置，在重载的 SessionFactory 函数中设定了定义实体的位置，这将促使定义的实体被作为域对象导入，RemoteServer 设定连接 Neo4j 服务器的 URL、用户名和密码，这些参数要依据安装 Neo4j 服务器的情况来设置。如果还没有安装 Neo4j 服务器，可参考附录 A 的方法进行安装，安装完成后启动服务器以备使用。

代码清单 2-25 Neo4j 配置类

```
@Configuration
@EnableTransactionManagement
@EnableNeo4jRepositories(basePackages = { "dbdemo.neo4j.repositories" })
public class Neo4jConfig extends Neo4jConfiguration {
    @Override
    public Neo4jServer neo4jServer() {
        return new RemoteServer("http://192.168.1.221:7474","neo4j","12345678");
    }

    @Override
```

```
    public SessionFactory getSessionFactory() {
        return new SessionFactory("dbdemo.neo4j.domain");
    }
}
```

现在可以编写一个测试程序来验证和演示上面编写的代码的功能，如代码清单 2-26 所示。这个测试程序分别创建了三部电影和三个演员，以及三个演员在三部电影中各自扮演的角色，然后按照电影标题查出一部电影，按照其内在的关系输出这部电影的信息和每个演员扮演的角色。这些数据的内容参照了 Neo4j 帮助文档中提供的示例数据。

代码清单 2-26　使用 Neo4j 的 JUint 测试程序

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {Neo4jConfig.class})
public class MovieTest {
    private static Logger logger = LoggerFactory.getLogger(MovieTest.class);

    @Autowired
    MovieRepository movieRepository;

    @Before
    public void initData(){
        movieRepository.deleteAll();

        Movie matrix1 = new Movie();
        matrix1.setTitle("The Matrix");
        matrix1.setYear("1999-03-31");

        Movie matrix2 = new Movie();
        matrix2.setTitle("The Matrix Reloaded");
        matrix2.setYear("2003-05-07");

        Movie matrix3 = new Movie();
        matrix3.setTitle("The Matrix Revolutions");
        matrix3.setYear("2003-10-27");

        Actor keanu = new Actor();
        keanu.setName("Keanu Reeves");

        Actor laurence = new Actor();
        laurence.setName("Laurence Fishburne");

        Actor carrieanne = new Actor();
```

```
        carrieanne.setName("Carrie-Anne Moss");

        matrix1.addRole(keanu,  "Neo");
        matrix1.addRole(laurence, "Morpheus");
        matrix1.addRole(carrieanne,  "Trinity");
        movieRepository.save(matrix1);
        Assert.notNull(matrix1.getId());

        matrix2.addRole(keanu, "Neo");
        matrix2.addRole(laurence, "Morpheus");
        matrix2.addRole(carrieanne,  "Trinity");
        movieRepository.save(matrix2);
        Assert.notNull(matrix2.getId());

        matrix3.addRole(keanu, "Neo");
        matrix3.addRole(laurence, "Morpheus");
        matrix3.addRole(carrieanne, "Trinity");
        movieRepository.save(matrix3);
        Assert.notNull(matrix3.getId());
    }

    @Test
    public void get(){
        Movie movie = movieRepository.findByTitle("The Matrix");
        Assert.notNull(movie);
        logger.info("===movie=== movie:{}, {}",movie.getTitle(), movie.getYear());
        for(Role role : movie.getRoles()){
            logger.info("====== actor:{}, role:{}", role.getActor().getName(),
role.getRole());
        }
    }
}
```

在 IDEA 的 Run/Debug Configuration 配置中增加一个 JUint 的配置项目，模块选择 neo4j，工作目录选择模块所在的根目录，测试程序选择 MovieTest 这个类，并将配置保存为 neo4jtest。

使用 Debug 模式运行测试项目 neo4jtest，如果测试通过，将在控制台中看到输出查询的这部电影和所有演员及其扮演的角色，如下所示：

```
=== movie=== movie:The Matrix, 1999-03-31
====== actor:Keanu Reeves, role:Neo
====== actor:Laurence Fishburne, role:Morpheus
====== actor:Carrie-Anne Moss, role:Trinity
```

这时，在数据库客户端的控制台上，单击左面侧边栏的关系类型 ACTS_IN，可以看到一个很酷的图形，图中每部电影和每个演员是一个节点，节点的每条有向边代表了这个演员在那部电影中扮演的角色，如图 2-7 所示。

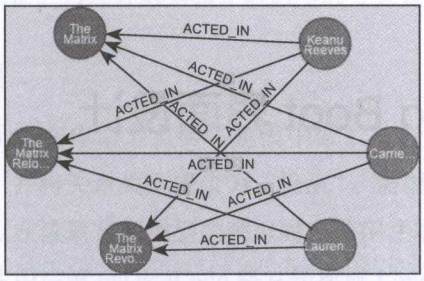


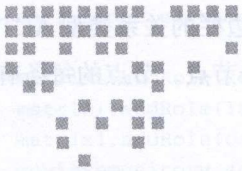
图 2-7　演员和电影的角色关系图

2.5　小结

这一章，我们一口气学习使用了 4 种数据库：MySQL、Redis、MongoDB、Neo4j，除了 Redis 以外，都使用了由 Spring Boot 提供的资源库来访问数据库并对数据库执行了一般的存取操作。可以看出，在 Spring Boot 框架中使用数据库非常简单、容易，这主要得益于 Spring Boot 资源库的强大功能，Spring Boot 资源库整合了第三方资源，它把复杂的操作变成简单的调用，它把所有“辛苦、繁重的事情”都包揽了，然后将“微笑和鲜花”献给了我们。为此，我们应该说声谢谢，谢谢开发 Spring Boot 框架及所有第三方提供者的程序员们，因为有了他们辛勤的付出，才有了我们今天使用上的便利。

本章实例的完整代码可以在 IDEA 中直接从 GitHub 中检出：https://github.com/chenfromsz/spring-boot-db.git。

本章实例都是使用 JUint 的方式来验证的，为了能使用友好的界面来运行应用，下一章将介绍如何使用 Thymeleaf 来进行界面设计。



Chapter 3

第 3 章

Spring Boot 界面设计

用 Spring Boot 框架设计 Web 显示界面，我们还是使用 MVC(Model View Controller，模型 – 视图 – 控制器）的概念，将数据管理、事件控制和界面显示进行分层处理，实现多层结构设计。界面设计，即视图的设计，主要是组织和处理显示的内容，界面上的事件响应最终交给了控制器进行处理，由控制器决定是否调用模型进行数据的存取操作，然后再将结果返回给合适的视图显示。

本章的实例工程使用分模块管理，如表 3-1 所示，即将数据管理独立成为一个工程模块，专门负责数据库管理方面的功能。而界面设计模块主要负责控制器和视图设计方面的功能。

表 3-1　实例工程模块列表

项　目	工　程	功　能
数据管理模块	data	实现使用 Neo4j 数据库
界面设计模块	webui	控制器和视图设计

3.1　模型设计

数据管理模块实现了 MVC 中模型的设计，主要负责实体建模和数据库持久化等方面的功能。在本章的实例中，将使用上一章的 Neo4j 数据库的例子，对电影数据进行管理。

回顾一下，有两个节点实体（电影和演员）和一个关系实体（角色）。其中，关系实体体现了节点实体之间的关系，即一个演员在一部电影中扮演一个角色。实体建模和持久化与上一章的实现差不多。只不过为了适应本章的内容，电影节点实体和角色关系实体的建模在属性上做了些许调整。另外针对 Neo4j 数据库的分页查询也做了一些调整和优化。

3.1.1　节点实体建模

如代码清单 3-1 所示，在电影节点实体建模中做了一些调整，即增加一个 photo 属性，用来存放电影剧照，并将关系类型更改为“扮演”。需要注意的是，Neo4j 还没有日期格式的数据类型，所以在读取日期类型的数据时，使用注解 @DateTimeFormat 进行格式转换，而在保存时，使用注解 @DateLong 将它转换成 Long 类型的数据进行存储。

代码清单 3-1　电影节点实体建模

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@NodeEntity
public class Movie {
    @GraphId
    Long id;
    private String name;
    private String photo;
    @DateLong
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date createDate;

    @Relationship(type="扮演", direction = Relationship.INCOMING)
    List<Role> roles = new ArrayList<>();

    public Role addRole(Actor actor, String name){
        Role role = new Role(actor,this,name);
        this.roles.add(role);
        return role;
    }

    public Movie() { }
    ......
}
```

3.1.2　关系实体建模

电影实体对应的角色关系实体建模的关系类型也同样做了调整而改为“扮演”，如

代码清单 3-2 所示。

代码清单 3-2 角色关系实体建模

```
@JsonIdentityInfo(generator=JSOGGenerator.class)
@RelationshipEntity(type = "扮演")
public class Role {
    @GraphId
    Long id;
    String name;
    @StartNode
    Actor actor;
    @EndNode
    Movie movie;

    public Role() {
}
......
}
```

3.1.3 分页查询设计

对于新型的 Neo4j 数据库来说，由于它的资源库遵循了 JPA 的规范标准来设计，在分页查询方面有的地方还不是很完善，所以在分页查询中，设计了一个服务类来处理，如代码清单 3-3 所示。其中，使用 Class<T> 传入调用的实体对象，使用 Pageable 传入页数设定和排序字段设定的参数，使用 Filters 传入查询的一些节点属性设定的参数。

代码清单 3-3 Neo4j 分页查询服务类

```
@Service
public class PagesService<T> {
    @Autowired
    private Session session;

    public Page<T> findAll(Class<T> clazz, Pageable pageable, Filters filters){
        Collection data = this.session.loadAll(clazz, filters, convert
(pageable.getSort()), new Pagination(pageable.getPageNumber(), pageable.
getPageSize()), 1);
        return updatePage(pageable, new ArrayList(data));
    }
.....
```

3.2　控制器设计

怎样将视图上的操作与模型——数据管理模块联系起来，这中间始终是控制器在起着通信桥梁的作用，它响应视图上的操作事件，然后根据需要决定是否访问数据管理模块，最后再将结果返回给合适的视图，由视图处理显示。下面将按照电影控制器的设计来说明控制器中增删查改的实现方法，演员控制器的设计与此类似，不再赘述。

3.2.1　新建控制器

接收新建电影的请求，以及输入一部电影的数据后的最后提交，由新建控制器进行处理。在控制器上将执行两个操作，第一个操作将返回一个新建电影的视图，第二个操作接收界面中的输入数据，并调用数据管理模块进行保存，如代码清单 3-4 所示。其中，create 函数将返回一个新建电影的视图，它不调用数据管理模块，save 函数将需要保存的数据通过调用数据管理模块存储至数据库中，并返回一个成功标志。注意，为了简化设计，将电影剧照的图片文件做了预定义处理。

代码清单 3-4　新建电影控制器

```
@RequestMapping("/new")
    public ModelAndView create(ModelMap model){
        String[] files = {"/images/movie/ 西游记 .jpg","/images/movie/ 西游记续集 .jpg"};
        model.addAttribute("files",files);
        return new ModelAndView("movie/new");
    }

    @RequestMapping(value="/save", method = RequestMethod.POST)
    public String save(Movie movie) throws Exception{
        movieRepository.save(movie);
        logger.info(" 新增 ->ID={}", movie.getId());
        return "1";
    }
```

3.2.2　查看控制器

查看一个电影的详细信息时，控制器首先使用请求的电影 ID 向数据管理模块请求数据，然后将取得的数据输出到一个显示视图上，如代码清单 3-5 所示。

代码清单 3-5 查看电影控制器

```
@RequestMapping(value="/{id}")
    public ModelAndView show(ModelMap model, @PathVariable Long id) {
        Movie movie = movieRepository.findOne(id);
        model.addAttribute("movie",movie);
        return new ModelAndView("movie/show");
}
```

3.2.3 修改控制器

若要实现对电影的修改及保存操作，需要先将电影的数据展示在视图界面上，然后接收界面的操作，调用数据管理模块将更改的数据保存至数据库中，如代码清单 3-6 所示。其中，为了简化设计，将剧照中的图片文件和电影角色名称做了预定义处理。修改数据时，由于从界面传回的电影对象中，丢失了其角色关系的数据（这是 OGM 的缺点），所以再次查询一次数据库，以取得一个电影的完整数据，然后再执行修改的操作。

代码清单 3-6 修改电影控制器

```
@RequestMapping(value="/edit/{id}")
    public ModelAndView update(ModelMap model, @PathVariable Long id){
        Movie movie = movieRepository.findOne(id);
        String[] files = {"/images/movie/ 西游记 .jpg","/images/movie/ 西游记续集 .jpg"};
        String[] rolelist = {" 唐僧 "," 孙悟空 "," 猪八戒 "," 沙僧 "};
        Iterable<Actor> actors = actorRepository.findAll();

        model.addAttribute("files",files);
        model.addAttribute("rolelist",rolelist);
        model.addAttribute("movie",movie);
        model.addAttribute("actors",actors);

        return new ModelAndView("movie/edit");
    }

    @RequestMapping(method = RequestMethod.POST, value="/update")
    public String update(Movie movie, HttpServletRequest request) throws Exception{
        String rolename = request.getParameter("rolename");
        String actorid = request.getParameter("actorid");

        Movie old = movieRepository.findOne(movie.getId());
        old.setName(movie.getName());
        old.setPhoto(movie.getPhoto());
        old.setCreateDate(movie.getCreateDate());
```

```
        if(!StringUtils.isEmpty(rolename) && !StringUtils.isEmpty(actorid)) {
            Actor actor = actorRepository.findOne(new Long(actorid));
            old.addRole(actor, rolename);
        }
        movieRepository.save(old);
        logger.info(" 修改 ->ID="+old.getId());
        return "1";
    }
```

3.2.4　删除控制器

删除电影时，从界面上接收电影的 ID 参数，然后调用数据管理模块将电影删除，如代码清单 3-7 所示。

代码清单 3-7　删除电影控制器

```
    @RequestMapping(value="/delete/{id}",method = RequestMethod.GET)
        public String delete(@PathVariable Long id) throws Exception{
            Movie movie = movieRepository.findOne(id);
            movieRepository.delete(movie);
            logger.info(" 删除 ->ID="+id);
            return "1";
    }
```

3.2.5　分页查询控制器

列表数据的查询使用分页的方法，按提供的查询字段参数、页码、页大小及其排序字段等参数，通过调用数据管理模块进行查询，然后返回一个分页对象 Page，如代码清单 3-8 所示。这里的分页查询调用了 3.1.3 节定义的分页查询服务类。

代码清单 3-8　电影分页查询控制器

```
        @RequestMapping(value="/list")
    public Page<Movie> list(HttpServletRequest request) throws Exception{
        String name = request.getParameter("name");
        String page = request.getParameter("page");
        String size = request.getParameter("size");
        Pageable pageable = new PageRequest(page==null? 0: Integer.parseInt(page),
size==null? 10:Integer.parseInt(size),
                new Sort(Sort.Direction.DESC, "id"));

        Filters filters = new Filters();
        if (!StringUtils.isEmpty(name)) {
```

```
            Filter filter = new Filter("name", name);
            filters.add(filter);
        }

        return pagesService.findAll(Movie.class, pageable, filters);
    }
```

3.3 使用 Thymeleaf 模板

完成了模型和控制器的设计之后，接下来的工作就是视图设计了。在视图设计中主要使用 Thymeleaf 模板来实现。在进行视图设计之前，先了解一下 Thymeleaf 模板的功能。

Thymeleaf 是一个优秀的面向 Java 的 XML/XHTML/HTML 5 页面模板，并具有丰富的标签语言和函数。使用 Spring Boot 框架进行界面设计，一般都会选择 Thymeleaf 模板。

3.3.1 Thymeleaf 配置

要使用 Thymeleaf 模板，首先，必须在工程的 Maven 管理中引入它的依赖："spring-boot-starter-thymeleaf"，如代码清单 3-9 所示。

代码清单 3-9 Thymeleaf 依赖配置

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

其次，必须配置使用 Thymeleaf 模板的一些参数。在一般的 Web 项目中都会使用如代码清单 3-10 所示的配置，其中，prefix 指定了 HTML 文件存放在 webapp 的 /WEB-INF/views/ 目录下面，或者也可以指定其他路径，其他一些参数的设置其实是使用了 Thymeleaf 的默认设置。

在实例中，为了更方便将项目发布成 jar 文件，我们将使用 Thymeleaf 自动配置中的默认配置选项，即只要在资源文件夹 resoueces 中增加一个 templates 目录即可，这个目录用来存放 HTML 文件。

代码清单 3-10 Thymeleaf 配置

```
spring:
    thymeleaf:
```

```
prefix: /WEB-INF/views/
suffix: .html
mode: HTML5
encoding: UTF-8
content-type: text/html
cache: false
```


注意 如果工程中增加了Thymeleaf的依赖，而没有进行任何配置，或者增加默认目录，启动应用时就会报错。

3.3.2 Thymeleaf功能简介

在HTML页面上使用Thymeleaf标签语言，用一个简单的关键字“th”来标注。使用Thymeleaf标签语言的典型例子如下：

```
<h3 th:text="${actor.name}"></h3>
<img th:src="@{/images/logo.png}"/>
```

其中，th:text指定了在标签<h3>中显示的文本，它的值来自于关键字“$”所引用的内存变量，th:src设定了标签<img>的图片文件的链接地址，既可以是绝对路径，也可以是相对路径。下面列出了Thymeleaf的一些主要标签和函数。

```
th:text，显示文本。
th:utext：和th:text的区别是针对"unescaped text"。
th:attr：设置标签属性。
th:if or th:unless：条件判断语句。
th:switch，th:case：选择语句。
th:each：循环语句。

#dates：日期函数。
#calendars：日历函数。
#numbers：数字函数。
#strings：字符串函数。
#objects：对象函数。
#bools：逻辑函数。
#arrays：数组函数。
#lists：列表函数。
```

本章的实例工程将在视图设计中使用Thymeleaf的下列几个主要功能，而有关Thymeleaf的详细说明和介绍可以访问它的官方网站http://www.thymeleaf.org/，以获得

更多的帮助。

1. 使用功能函数

Thymeleaf有一些日期功能函数、字符串函数、数组函数、列表函数等，代码清单3-11是Thymeleaf使用日期函数的一个例子，#dates.format是一个日期格式化的使用实例，它将电影的创建日期格式化为中文环境的使用格式“'yyyy-MM-dd HH:mm:ss'”。

代码清单3-11 Thymeleaf使用函数

```
th:value="${movie.createDate} ? ${#dates.format(movie.createDate,'yyyy-MM-
dd HH:mm:ss')} ':''"
```

2. 使用编程语句

Thymeleaf有条件语句、选择语句、循环语句等。代码清单3-12使用each循环语句来显示一个数据列表，即在下拉列表框中使用循环语句来显示所有的演员列表。

代码清单3-12 th:each循环

```
<select name="actorid" id="actorid">
<option value="">选择演员 </option>
<option th:each="actor:${actors}"
    th:value="${actor.id}"
    th:text="${actor.name}">
</option>
</select>
```

3. 使用页面框架模板

Thymeleaf的页面框架模板是比较优秀的功能。预先定义一个layout，它具有页眉、页脚、提示栏、导航栏和内容显示等区域，如代码清单3-13所示。其中，layout:fragment="prompt"是一个提示栏，它可以让引用的视图替换显示的内容；fragments/nav :: nav是一个导航栏并指定了视图文件，也就是说它不能被引用的视图替换内容；layout:fragment="content"是一个主要内容显示区域，它也能由引用的视图替换显示内容；fragments/footer :: footer是一个页脚定义并且也指定了视图文件，即不被引用的视图替换显示内容。这样设计出来的页面模板框架如图3-1所示。

代码清单3-13 layout模板

```
<div class="headerBox">
    <div class="topBox">
```

```
        <div class="topLogo f-left">
            <a href="#"><img th:src="@{/images/logo.png}"/></a>
        </div>
        <div class="new-nav">
            <h3> 电影频道 </h3>
        </div>
    </div>
</div>
<div class="locationLine" layout:fragment=" prompt ">
    当前位置: 首页 &gt; <em> 页面 </em>
</div>
<table class="globalMainBox" style="position:relative;z-index:1">
    <tr>
        <td class="columnLeftBox" valign="top">
            <div th:replace="fragments/nav :: nav"></div>
        </td>
        <td class="whiteSpace"></td>
        <td class="rightColumnBox" valign="top">
            <div layout:fragment="content"></div>
        </td>
    </tr>
</table>

<div class="footBox" th:replace="fragments/footer :: footer"></div>
```

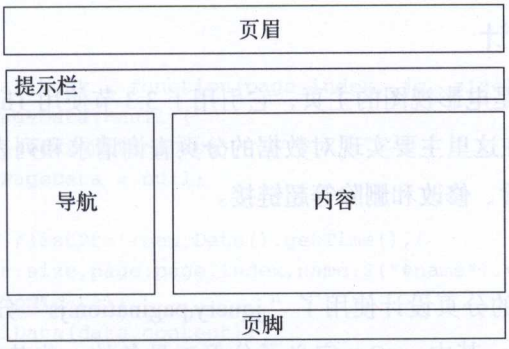


图 3-1　页面框架模板

有了页面模板之后，就可以在一个主页面视图上引用上面的 layout，并替换它的提示栏 prompt 和主要内容显示区域 content，其他页眉、页脚和导航栏却保持同样的内容，如代码清单 3-14 所示。这样就可以设计出一个使用共用模板的具有统一风格特征的界面。

代码清单 3-14 使用 layout 模板的视图设计

```
<html xmlns:th="http://www.thymeleaf.org" layout:decorator="fragments/layout">
......
<div class="locationLine" layout:fragment=" prompt ">
当前位置: 首页 &gt; <em > 电影管理 </em>
</div>

<div class="statisticBox w-782"  layout:fragment="content">
......
</div>
```

3.4 视图设计

视图设计包括列表视图、新建视图、查看视图、修改视图和删除视图设计等 5 个方面有关数据的增删查改的内容。

我们知道，视图上的数据存取不是直接与模型打交道，而是通过控制器来处理。在视图中对于控制器的请求，大多使用 jQuery 的方式来实现。jQuery 是一个优秀的 JavaScript 程序库，并且具有很好的兼容性，几乎兼容了现有的所有浏览器。

下面的视图设计将以电影的视图设计为例说明，演员的视图设计与此类似，不再赘述。

3.4.1 列表视图设计

电影的列表视图是电影视图的主页，它引用了 3.3 节使用 Thymeleaf 设计的页面框架模板 layout.html，在这里主要实现对数据的分页查询请求和列表数据显示，并提供了一部电影的新建、查看、修改和删除等超链接。

1. 分页设计

电影的列表视图的分页设计使用了“jquery.pagination.js”分页插件，编写如代码清单 3-15 所示的脚本，其中 getOpt 定义了分页工具条的一些基本属性，pageaction 通过“./list”调用控制器取得分页数据列表，fillData 函数将列表数据填充到 HTML 控件 tbodyContent 中。

代码清单 3-15 分页设计的 js 编码

```
// 分页的参数设置
var getOpt = function(){
    var opt = {
```

```
        items_per_page: 10,                    //每页记录数
        num_display_entries: 3,                //中间显示的页数，默认为10
        current_page:0,                        //当前页
        num_edge_entries:1,                    //头尾显示的页数，默认为0
        link_to:"javascript:void(0)",
        prev_text:"上页",
        next_text:"下页",
        load_first_page:true,
        show_total_info:true ,
        show_first_last:true,
        first_text:"首页",
        last_text:"尾页",
        hasSelect:false,
        callback: pageselectCallback    //回调函数
    }
    return opt;
}
//分页开始
var currentPageData = null ;
var pageaction = function(){
    $.get('./list?t='+new Date().getTime(),{
        name:$("#name").val()
    },function(data){
        currentPageData = data.content;
        $(".pagination").pagination(data.totalElements, getOpt());
    });
}

var pageselectCallback = function(page_index, jq, size){
    if(currentPageData!=null){
        fillData(currentPageData);
        currentPageData = null;
    }else
        $.get('./list?t='+new Date().getTime(),{
            size:size,page:page_index,name:$("#name").val()
        },function(data){
            fillData(data.content);
        });
}
//填充分页数据
function fillData(data){
    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v){
        var html = "" ;
        html += '<tr> ' +
                '<td>'+ (v.id==null?'':v.id) +'</td>' +
```

```
                '<td>'+ (v.name==null?'':v.name) +'</td>' +
                '<td>'+ (v.createDate==null?'': getSmpFormatDateByLong(v.create
Date,true)) +'</td>' ;
            html += '<td><a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"detail(\''+ v.id+'\')">查 看</a><a class="c-50a73f mlr-6" href=
"javascript:void(0)" onclick="edit(\''+ v.id+'\')">修 改</a><a class="c-50a73f
mlr-6" href="javascript:void(0)" onclick="del(\''+ v.id+'\')">删除</a></td>' ;
            html +='</tr>' ;

            $list.append($(html));
        });
    }
    //分页结束
```

2. 列表页面设计

电影列表的显示页面主要定义了列表字段的名称和提供显示数据内容的控件 ID，即 tbodyContent，如代码清单 3-16 所示。

代码清单 3-16 电影列表页面 HTML 编码

```
    <!DOCTYPE html>
    <html xmlns:th="http://www.thymeleaf.org" layout:decorator="fragments/
layout">
    <head>
        <title>电影管理</title>

        <link th:href="@{/scripts/pagination/pagination.css}" rel="stylesheet"
type="text/css" />
        <link th:href="@{/scripts/artDialog/default.css}" rel="stylesheet"
type="text/css" />
        <link th:href="@{/scripts/My97DatePicker/skin/WdatePicker.css}"
rel="stylesheet" type="text/css" />
        <link th:href="@{/styles/index.css}" rel="stylesheet"  type="text/css"/>

        <script th:src="@{/scripts/pagination/jquery.pagination.js}"></script>
        <script th:src="@{/scripts/jquery.smartselect-1.1.min.js}"></script>
        <script th:src="@{/scripts/My97DatePicker/WdatePicker.js}"></script>
        <script th:src="@{/scripts/movie/list.js}"></script>
    </head>
    <body>
    <div class="locationLine" layout:fragment="prompt">
        当前位置：首页 &gt; <em >电影管理</em>
    </div>

    <div class="statisticBox w-782"  layout:fragment="content">
```

```
        <form id="queryForm" method="get">
        <div class="radiusGrayBox782">
            <div class="radiusGrayTop782"></div>
            <div class="radiusGrayMid782">
                <div class="dataSearchBox forUserRadius">
                    <ul>
                        <li>
                            <label class="preInpTxt f-left">电影名称 </label>
                            <input type="text" class="inp-list f-left w-200" place
holder=" 按电影名称搜索 " id="name"  name="name"/>
                        </li>
                        <li>
                            <a href="javascript:void(0)" class="blueBtn-62X30 f-right"
id="searchBtn">查询 </a>
                        </li>
                    </ul>
                </div>
            </div>
        </div>
        </form>
        <div class="newBtnBox">
            <input type="hidden" id="m_ck" />
            <a id="addBtn" class="blueBtn-62X30" href="javascript:void(0)">新增 </a>
        </div>
        <div class="dataDetailList mt-12">
            <table id="results" class="dataListTab">
                <thead>
                <tr>
                    <th>ID</th>
                    <th> 电影 </th>
                    <th> 出版日期 </th>
                    <th> 操作 </th>
                </tr>
                </thead>
                <tbody id="tbodyContent">
                </tbody>
            </table>
            <div class="tableFootLine">
                <div class="pagebarList pagination"/>
            </div>
        </div>
    </div>

    </body>
    </html>
```

3. 列表视图设计效果

电影数据列表视图设计的最终显示效果如图 3-2 所示。

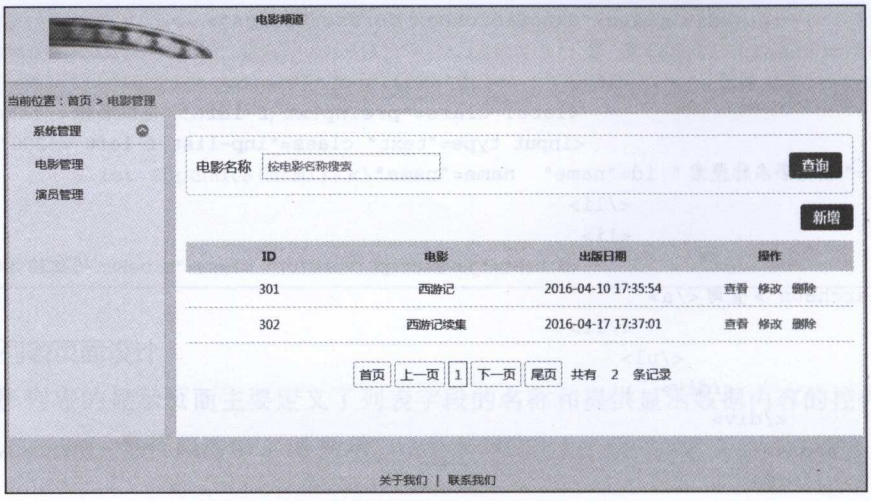


图 3-2 电影列表视图设计效果图

3.4.2 新建视图设计

1. 新建对话框设计

新建电影时，在电影主页中打开一个对话框显示新建的操作界面，对话框设计引用了“artDialog.js”的对话框插件，然后编写一个脚本来打开对话框，如代码清单 3-17 所示。其中“./new”是连接控制器的请求 URL，注意这里使用了相对路径，这个 URL 通过“$.get”请求返回新建电影的 HTML 页面，请求链接中的 ts 参数传递的是当前时间，这是为了保证该链接是一个全新的链接，以使浏览器能显示一个最新的内容页面。

代码清单 3-17 新建电影对话框设计 js 编码

```
function create(){
    $.get("./new",{ts:new Date().getTime()},function(data){
        art.dialog({
            lock:true,
            opacity:0.3,
            title: "新增",
            width:'750px',
            height: 'auto',
            left: '50%',
```

```
                top: '50%',
                content:data,
                esc: true,
                init: function(){
                    artdialog = this;
                },
                close: function(){
                    artdialog = null;
                }
            });
        });
    }
```

2. 新建页面设计

新建电影的页面设计，如代码清单 3-18 所示，这里只是部分 HTML 编码，其中的日期控件使用“ WdatePicker.js”插件来实现。对于一部电影来说，我们需要输入名称、剧照和日期三个属性，其中剧照的图片下拉列表框使用“ imageselect.js”图片下拉列表框插件来实现，并且为了简化设计，剧照中的图片文件使用了预先定义的文件，这里只要选择使用哪一个图片即可。

代码清单 3-18 新建电影页面 HTML 编码

```
    <link th:href="@{/styles/imageselect.css}" rel="stylesheet" type="text/css" />
    <script th:src="@{/scripts/imageselect.js}"></script>
    <script th:src="@{/scripts/movie/new.js}"></script>

    <form id="saveForm" action="./save" method="post">
        <table class="addNewInfList">
            <tr>
                <th>名称</th>
                <td><input class="inp-list w-200 clear-mr f-left" type="text"  id=
"name" name="name"  maxlength="120" /></td>
                <th>剧照</th>
                <td width="240">
                    <select name="photo" id="photo">
                        <option th:each="file:${files}"
                                th:value="${file}"
                                th:text="${file}">
                        </option>
                    </select>
                </td>
            </tr>
            <tr>
```

```
                <th> 日期 </th>
                <td>
                    <input  onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" id="createDate" name="createDate"/>
                </td>
            </tr>
        </table>
        <div class="bottomBtnBox">
            <a class="btn-93X38 saveBtn" href="javascript:void(0)"> 确定 </a>
            <a class="btn-93X38 backBtn" href="javascript: closeDialog()"> 返回 </a>
        </div>
    </form>
    <script type="text/javascript">
        $(document).ready(function(){
            $('select[name=photo]').ImageSelect({dropdownWidth:425});
        });
    </script>
```

3. 表单验证与提交设计

验证新建电影表单的提交时使用“jquery.validate.min.js”插件中的验证方法来实现，如代码清单 3-19 所示。保存时调用经典的“$.ajax”方式利用 POST 方法进行提交，其中 headers: {"Content-type": "application/x-www-form-urlencoded;charset=UTF-8"} 用于保证数据在传输过程中中文字符的正确性。在表单验证中，只对 name 和 createDate 两个属性进行简单的非空验证，表单的参数传递使用一个表单序列化函数 serialize() 来实现，它将表单控件上的对象序列化为一个个含有“键 – 值”对的字符串进行提交。

代码清单 3-19 新建电影中表单验证和提交的 js 编码

```
$(function(){
        $('#saveForm').validate({
                rules: {
            name        :{required:true},
            createDate      :{required:true}
                },messages:{
            name :{required:" 必填 "},
            createDate :{required:" 必填 "}
        }
        });
    $('.saveBtn').click(function(){
        if($('#saveForm').valid()){
            $.ajax({
```

```
                type: "POST",
                url: "./save",
                data: $("#saveForm").serialize(),
                headers: {"Content-type": "application/x-www-form-urlencoded;
charset=UTF-8"},
                success: function (data) {
                    if (data == 1) {
                        alert("保存成功");
                        pageaction();
                        closeDialog();
                    } else {
                        alert(data);
                    }
                },
                error:function(data){
                    var e;
                    $.each(data,function(v){
                        e += v + " ";
                    });
                    alert(e);
                }
            });
        }else{
            alert('数据验证失败，请检查！');
        }
    });
});
```

4. 新建视图设计效果

新建电影的视图设计最后的显示效果如图 3-3 所示。

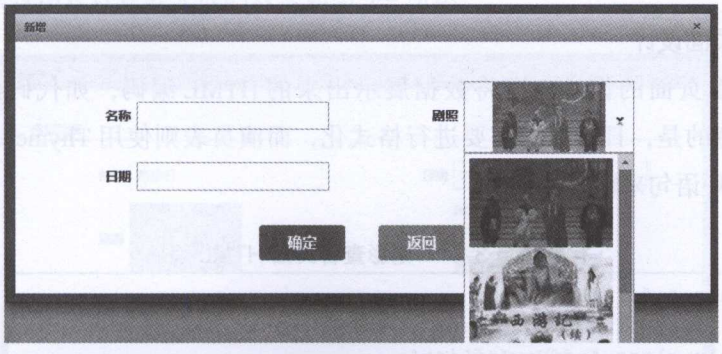


图 3-3　新建电影视图设计效果图

3.4.3 查看视图设计

1. 查看对话框设计

在电影的主页中单击一部电影的查看链接，将打开一个查看电影的对话框，对话框的设计如代码清单 3-20 所示，其中“./{id}”是提取数据的链接，它将向控制器请求数据，并以 HTML 页面方式显示出来。

代码清单 3-20 查看电影对话框 js 编码

```
function detail(id){
    $.get("./"+id,{ts:new Date().getTime()},function(data){
        art.dialog({
            lock:true,
            opacity:0.3,
            title: "查看信息",
            width:'750px',
            height: 'auto',
            left: '50%',
            top: '50%',
            content:data,
            esc: true,
            init: function(){
                artdialog = this;
            },
            close: function(){
                artdialog = null;
            }
        });
    });
}
```

2. 查看页面设计

电影查看页面的设计，即将数据展示出来的 HTML 编码，如代码清单 3-21 所示，需要注意的是，日期数据需要进行格式化，而演员表则使用 Thymeleaf 中的一个“th:each”循环语句来输出。

代码清单 3-21 电影查看页面 HTML 编码

```
<div class="addInfBtn">
    <h3 class="itemTit"><span>电影信息</span></h3>
    <table class="addNewInfList">
        <tr>
            <th>名称</th>
```

```
                <td width="240"><input class="inp-list w-200 clear-mr f-left" type=
"text" th:value="${movie.name}" id="name" name="name" maxlength="16" /></td>
                <th>日期</th>
                <td>
                    <input onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="${movie.createDate}
? ${#dates.format(movie.createDate,'yyyy-MM-dd HH:mm:ss')} :''" id="createDate"
name="createDate"/>
                </td>
            </tr>

            <tr>
                <th>剧照</th>
                <td width="240">
                    <img th:src="${movie.photo}"/>
                </td>
                <th>演员表</th>
                <td width="240">
                    <ul>
                        <li th:each="role:${movie.roles}" th:text="${role.actor.
name}+' 饰 '+${role.name}"></li>
                    </ul>
                </td>
            </tr>
        </table>
        <div class="bottomBtnBox">
            <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
        </div>
    </div>
```

3. 查看视图的设计效果

电影查看视图设计最终完成的效果如图 3-4 所示。

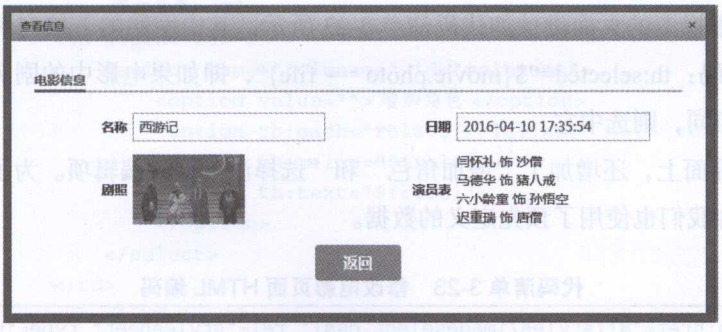


图 3-4　查看电影视图设计效果图

3.4.4 修改视图设计

1. 修改对话框设计

在电影的主页中修改一部电影，首先打开一个修改电影的对话框，这个对话框的设计如代码清单 3-22 所示。其中通过“ $.get”访问“ ./edit/{id}”取得数据和修改视图的 HTML 页面元素。

代码清单 3-22 修改电影对话框 js 编码

```
function edit(id){
    $.get("./edit/"+id,{ts:new Date().getTime()},function(data){
        art.dialog({
            lock:true,
            opacity:0.3,
            title: "修改",
            width:'750px',
            height: 'auto',
            left: '50%',
            top: '50%',
            content:data,
            esc: true,
            init: function(){
                artdialog = this;
            },
            close: function(){
                artdialog = null;
            }
        });
    });
}
```

2. 修改页面设计

修改电影视图的页面设计如代码清单 3-23 所示，其中剧照的下拉列表框中增加了“选中”的代码：th:selected="${movie.photo == file}"，即如果电影中的剧照与下拉框列表中的剧照相同，则选中它。

在修改界面上，还增加了“增加角色”和“选择演员”的编辑项。为了简化设计这里的角色名称我们也使用了预先定义的数据。

代码清单 3-23 修改电影页面 HTML 编码

```
<link th:href="@{/styles/imageselect.css}" rel="stylesheet" type="text/css" />
<script th:src="@{/scripts/imageselect.js}"></script>
```

```
<script th:src="@{/scripts/movie/edit.js}"></script>

<form id="saveForm" method="post">
    <input type="hidden" name="id" id="id" th:value="${movie.id}"/>
<div class="addInfBtn" >
    <h3 class="itemTit"><span>编辑信息 </span></h3>
    <table class="addNewInfList">
        <tr>
            <th>电影名称 </th>
            <td width="240"><input class="inp-list w-200 clear-mr f-left" type=
"text" th:value="${movie.name}" id="name" name="name" maxlength="16" /></td>
            <th>电影剧照 </th>
            <td width="240">
                <select name="photo" id="photo">
                    <option th:each="file:${files}"
                            th:value="${file}"
                            th:text="${file}"
                            th:selected="${movie.photo == file}">
                    </option>
                </select>
            </td>
        </tr>

        <tr>
            <th>出版日期 </th>
            <td>
                <input  onfocus="WdatePicker({dateFmt:'yyyy-MM-dd HH:mm:ss'})"
type="text" class="inp-list w-200 clear-mr f-left" th:value="${movie.
createDate} ? ${#dates.format(movie.createDate,'yyyy-MM-dd HH:mm:ss')} :''"
id="createDate" name="createDate"/>
            </td>

        </tr>
        <tr>
            <th>增加角色 </th>
            <td width="240">
                <select name="rolename" id="rolename">
                    <option value="">增加角色 </option>
                    <option th:each="role:${rolelist}"
                            th:value="${role}"
                            th:text="${role}">
                    </option>
                </select>
            </td>
            <th>选择演员 </th>
            <td width="240">
                <select name="actorid" id="actorid">
```

```
                    <option value="">选择演员</option>
                    <option th:each="actor:${actors}"
                            th:value="${actor.id}"
                            th:text="${actor.name}">
                    </option>
                </select>
            </td>
        </tr>
    </table>

    <div class="bottomBtnBox">
        <a class="btn-93X38 saveBtn" href="javascript:void(0)">确定</a>
        <a class="btn-93X38 backBtn" href="javascript:closeDialog(0)">返回</a>
    </div>
</div>

</form>
<script type="text/javascript">
    $(document).ready(function(){
        $('select[name=photo]').ImageSelect({dropdownWidth:425});
    });
</script>
```

3. 修改视图的设计效果

最终完成的修改电影视图的显示效果如图 3-5 所示。

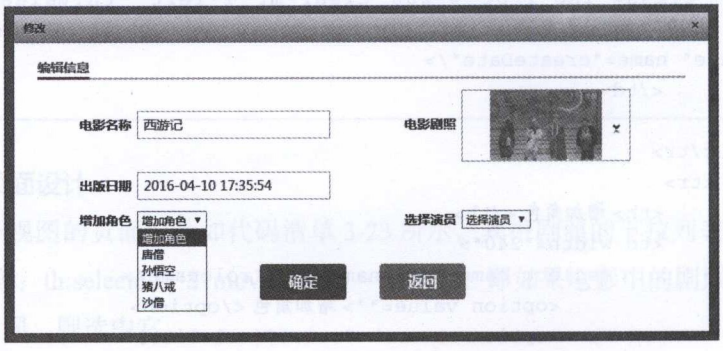


图 3-5 修改电影视图设计效果图

3.4.5 删除视图设计

1. 删除确认对话框

如果有删除的操作，首先要给出确认提示框，只有用户单击确定后才能删除数

据，否则将不做任何操作。确认提示框是调用了 Windows 中的确认对话框，如代码清单 3-24 所示。

代码清单 3-24　删除确认对话框 js 编码

```
function del(id){
    if(!confirm(" 您确定删除此记录吗？ ")){
        return false;
    }
    $.get("./delete/"+id,{ts:new Date().getTime()},function(data){
        if(data==1){
            alert(" 删除成功 ");
            pageaction();
        }else{
            alert(data);
        }
    });
}
```

2. 删除确认设计效果

执行删除操作的确认效果如图 3-6 所示。

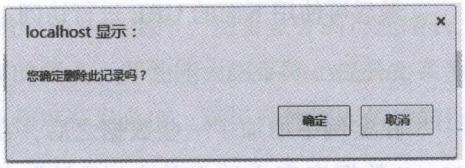


图 3-6　删除电影确认对话框效果图

3.5　运行与发布

本章实例工程的完整代码可以通过 IDEA 从 GitHub 中检出：https://github.com/chenfromsz/spring-boot-ui.git。Spring Boot 需要一个启动程序作为应用的入口，在 webui 模块中，我们设计了一个入口程序，如代码清单 3-25 所示。使用这个入口程序，就可以调试和发布工程了。

代码清单 3-25　Web 应用启动主程序

```
package com.test.webui;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan(basePackages = "com.test")
public class WebuiApp {
    public static void main(String[] args) {
        SpringApplication.run(WebuiApp.class, args);
```

```
    }
}
```

通过在 IDEA 中打开 Run/Debug Configurations 对话框，增加一个 Spring Boot 配置，模块选择 webui，工作目录选择模块 webui 所在的路径，主程序选择 WebuiApp，并将配置保存为 webui。然后在 IDEA 中运行该配置项目 webui，即可启动应用进行调试。

如果要发布应用，可以在 IDEA 的 Run/Debug Configurations 对话框中增加一个 Maven 打包配置项目，工作目录选择工程的根目录，命令行中输入指令：clean package-D skipTests，并将配置保存为 mvn。然后运行这个配置项目 mvn 进行打包，打包成功后，在“webui/target”目录中将生成 webui-1.0-SNAPSHOT.jar。要运行这个程序包，可以打开一个命令行窗口，将路径切换到 webui-1.0-SNAPSHOT.jar 所在的目录，使用下列指令即可运行应用。

```
java -jar webui-1.0-SNAPSHOT.jar
```

最后可使用下面的 URL 进行访问：

```
http://localhost
```

在实例中增加了一些数据之后，在 Neo4j 数据库客户端中单击“扮演”关系，也可以看到电影和演员的关系图，如图 3-7 所示。

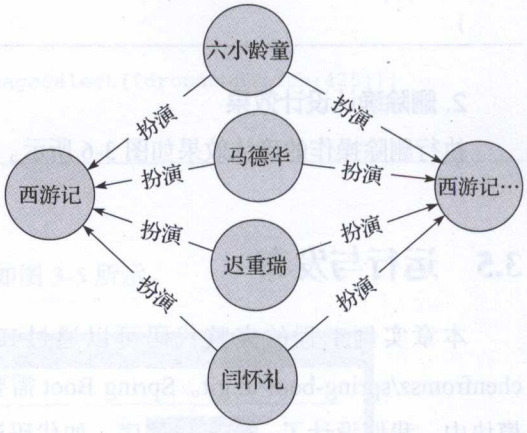
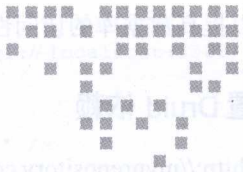


图 3-7 电影与演员关系图

3.6 小结

本章介绍了使用 MVC 的多层结构方式，以及在 Spring Boot 进行 Web 界面设计的方法，并且使用 Thymeleaf 模板设计了一个 Web 应用的页面框架。Web 界面设计的一些细节，更多的是使用了 HTML 编码和 JavaScript 脚本，而 HTML 离不开 CSS 的支持，JavaScript 更是借助于 jQuery 及其各种插件的功能。读者如需深入了解这方面的知识和技术，可查找相关的知识进行学习和研究。这里主要使用 Thymeleaf 模板工具来设计整体界面以及组织和处理数据的显示。

有了显示界面之后，对数据库的操作就更为方便和直观了。下一章将介绍如何使用一些技术手段来提升访问数据库的性能，以及怎样扩展访问数据库的功能。



第 4 章 Chapter 4

提高数据库访问性能

使用关系型数据库的应用系统的性能瓶颈最终还是数据库。随着业务的迅速增长，数据量会不断增大，会逐渐暴露出关系型数据库的弱点，即性能大幅下降。提升关系型数据库的访问性能是开发者的迫切任务。下面从程序开发角度，对提升数据库的访问性进行介绍和探讨。

本章的实例工程使用了分模块的方式设计，各个模块的功能如表 4-1 所示。

表 4-1 实例工程模块列表

项　目	工　程	类　型	功　能
扩展功能模块	dpexpan	程序集成	JPA 功能扩展和 Redis 配置等
数据管理模块	mysql	程序集成	MySQL 实体建模和持久化等
Web 应用模块	website	Web 应用	Web 应用实例

4.1 使用 Druid

Druid 是一个关系型数据库连接池，它是阿里巴巴的一个开源项目。Druid 支持所有 JDBC 兼容的数据库，包括 Oracle、MySQL、Derby、PostgreSQL、SQL Server、H2 等。Druid 在监控、可扩展性、稳定性和性能方面具有明显的优势。通过 Druid 提供的监控功能，可以实时观察数据库连接池和 SQL 查询的工作情况。使用 Druid 连接池，在一

定程度上可以提高数据库的访问性能。

4.1.1 配置 Druid 依赖

可以从 http://mvnrepository.com/ 中查找 Druid 的依赖配置，找到合适的版本，然后复制其中 Maven 的配置到实例工程的扩展功能模块 dpexpan 中。图 4-1 是我们查到的结果，使用的是 1.0.18 版本。图 4-1 中的 HomePage 是 Druid 的源代码链接地址。

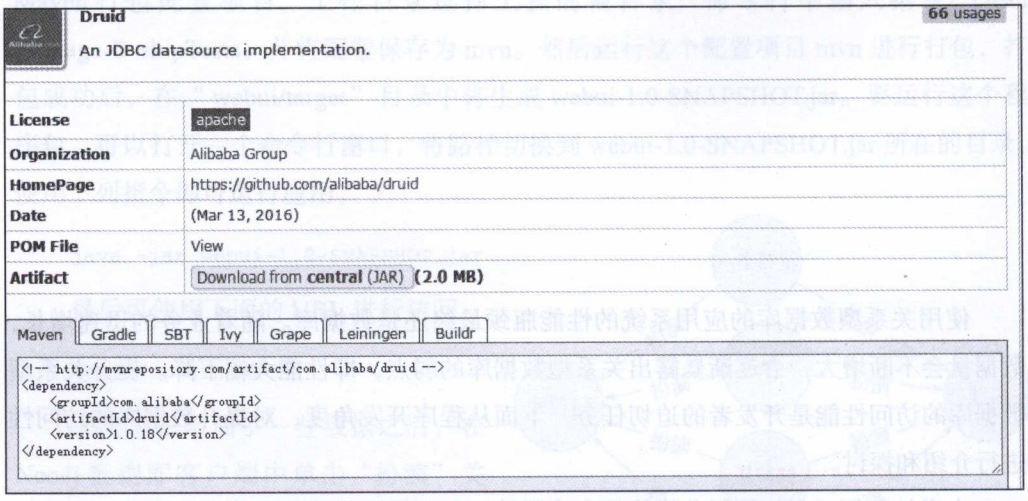


图 4-1 查找 Druid 的依赖配置

4.1.2 关于 XML 配置

使用 Spring 开发框架时，XML 配置是经常使用的一种配置方法，其中数据源配置就是使用 XML 配置中的一种。代码清单 4-1 是一个使用 Druid 连接池的 XML 配置。使用 Spring Boot 框架也能使用 XML 配置，只要在程序入口使用一个注解，如 @ImportResource ({"classpath:spring-datasource.xml"})，即可导入 XML 配置。但是，Spring Boot 不推荐这样使用，而是集中在配置文件 application.properties 或 application.yml 中进行配置。

代码清单 4-1 XML 数据源配置

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-
method="init" destroy-method="close">
        <!-- 驱动名称 -->
        <property name="DriverClassName" value="com.mysql.jdbc.Driver" />
```

```
        <!-- JDBC 连接串 -->
        <property name="url" value="jdbc:mysql:// localhost:3306/test?useUni
code=true&amp;characterEncoding=utf-8" />
        <!-- 数据库用户名称 -->
        <property name="username" value="root" />
        <!-- 数据库密码 -->
        <property name="password" value="12345678" />
        <!-- 连接池最大使用连接数量 -->
        <property name="maxActive" value="20" />
        <!-- 初始化大小 -->
        <property name="initialSize" value="5" />
        <!-- 获取连接最大等待时间 -->
        <property name="maxWait" value="60000" />
        <!-- 连接池最小空闲 -->
        <property name="minIdle" value="2" />
        <!-- 逐出连接的检测时间间隔 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <!-- 最小逐出时间 -->
        <property name="minEvictableIdleTimeMillis" value="300000" />
        <!-- 测试有效用的 SQL Query -->
        <property name="validationQuery" value="SELECT 'x'" />
        <!-- 连接空闲时测试是否有效 -->
        <property name="testWhileIdle" value="true" />
        <!-- 获取连接时测试是否有效 -->
        <property name="testOnBorrow" value="false" />
        <!-- 归还连接时是否测试有效 -->
        <property name="testOnReturn" value="false" />
        <!-- 配置监控统计拦截的 filters -->
        <property name="filters" value="stat" />
    </bean>
```

4.1.3　Druid 数据源配置

Spring Boot 的数据源配置的默认类型是 org.apache.tomcat.jdbc.pool.DataSource，为了使用 Druid 连接池，可以将数据源类型更改为 com.alibaba.druid.pool.DruidDataSource，如代码清单 4-2 所示。其中，url、username、password 是连接 MySQL 服务器的配置参数，其他一些参数设定 Druid 的工作方式。

代码清单 4-2　Druid 数据源配置

```
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
```

```
url: jdbc:mysql:// localhost:3306/test?characterEncoding=utf8
username: root
password: 12345678
# 初始化大小，最小，最大
initialSize: 5
minIdle: 5
maxActive: 20
# 配置获取连接等待超时的时间
maxWait: 60000
# 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
timeBetweenEvictionRunsMillis: 60000
# 配置一个连接在池中的最小生存时间，单位是毫秒
minEvictableIdleTimeMillis: 300000
validationQuery: SELECT 1 FROM DUAL
testWhileIdle: true
testOnBorrow: false
testOnReturn: false
# 打开 PSCache，并且指定每个连接上 PSCache 的大小
poolPreparedStatements: true
maxPoolPreparedStatementPerConnectionSize: 20
# 配置监控统计拦截的 filters，去掉后监控界面 sql 将无法统计，'wall' 用于防火墙
filters: stat,wall,log4j
# 通过 connectProperties 属性来打开 mergeSql 功能；慢 SQL 记录
connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
# 合并多个 DruidDataSource 的监控数据
#useGlobalDataSourceStat=true
```

上面配置中的 filters: stat 表示已经可以使用监控过滤器，这时结合定义一个过滤器，就可以用来监控数据库的使用情况。

注意 在 Spring Boot 低版本的数据源配置中，是没有提供设定数据源类型这一功能的，这时如果要使用上面这种配置方式，就需要使用自定义的配置参数来实现。

4.1.4 开启监控功能

开启 Druid 的监控功能，可以在应用运行的过程中，通过监控提供的多维度数据来分析使用数据库的运行情况，从而可以调整程序设计，以优化数据库的访问性能。

代码清单 4-3 定义了一个监控服务器和一个过滤器，监控服务器设定了访问监控后台的连接地址为“/druid/*”，设定了访问数据库的白名单和黑名单，即通过访问者的 IP 地址来控制访问来源，增加了数据库的安全设置，还配置了一个用来登录监控后台的用户 druid，并将密码设置为 12345678。

代码清单 4-3　开启 Druid 监控功能

```
@Configuration
public class DruidConfiguration {
    @Bean
    public ServletRegistrationBean statViewServle(){
        ServletRegistrationBean servletRegistrationBean = new ServletRegist
rationBean(new StatViewServlet(),"/druid/*");
        //IP 白名单
        servletRegistrationBean.addInitParameter("allow","192.168.1.218,127.
0.0.1");
        //IP 黑名单 (共同存在时，deny 优先于 allow)
        servletRegistrationBean.addInitParameter("deny","192.168.1.100");
        //控制台管理用户
        servletRegistrationBean.addInitParameter("loginUsername","druid");
        servletRegistrationBean.addInitParameter("loginPassword","12345678");
        //是否能够重置数据
        servletRegistrationBean.addInitParameter("resetEnable","false");
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean statFilter(){
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean
(new WebStatFilter());
        //添加过滤规则
        filterRegistrationBean.addUrlPatterns("/*");
        //忽略过滤的格式
        filterRegistrationBean.addInitParameter("exclusions","*.js,*.gif,*.jpg,*.
png,*.css,*.ico,/druid/*");
        return filterRegistrationBean;
        }
}
```

开启监控功能后，运行应用时，可以通过网址 http://localhost/durid/index.html 打开控制台，输入上面程序设定的用户名和密码，登录进去，可以打开如图 4-2 所示的监控后台。

在监控后台中，可以实时查看数据库连接池的情况，每一个被执行的 SQL 语句使

用的次数和花费的时间、并发数等，以及一个 URI 请求的次数、时间和并发数等情况。这就为分析一个应用系统访问数据库的情况和性能提供了可靠、详细的原始数据，让我们能在一些基础的细节上修改和优化一个应用访问数据库的设计。

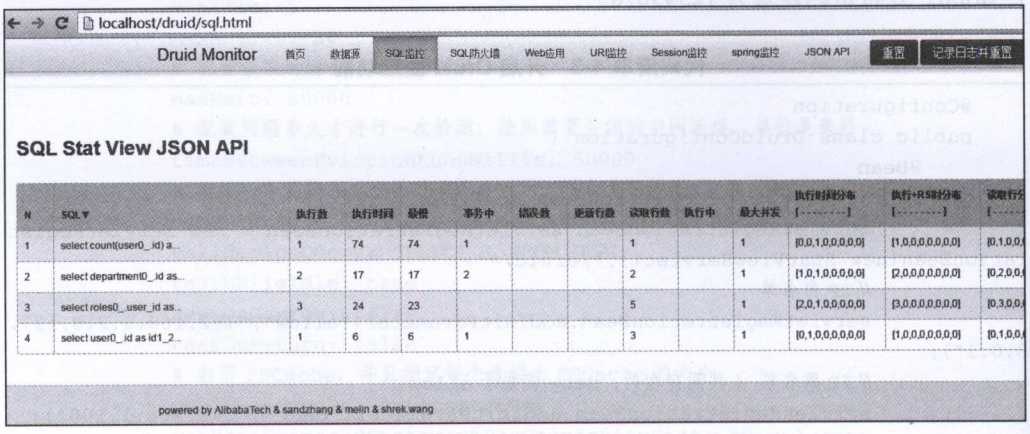


图 4-2 Druid 监控后台

如果需要了解更多有关 Druid 的使用或者下载其源代码，可以访问 https://github.com/alibaba/druid。

4.2 扩展 JPA 功能

使用 JPA，在资源库接口定义中不但可以按照其规则约定的方法声明各种方法（像在第 2 章中介绍的那样），也可以使用注解 @Query 来定义一些简单的查询语句，如代码清单 4-4 所示。本节还将介绍如何在全局的范围中，扩展 JPA 接口的功能。

代码清单 4-4 使用 @Query 自定义查询

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select t from User t where t.name =?1 and t.email =?2")
    User findByNameAndEmail(String name, String email);

    @Query("select t from User t where t.name like :name")
    Page<User> findByName(@Param("name") String name, Pageable pageRequest);
}
```

4.2.1　扩展 JPA 接口

首先创建一个接口，继承于 JpaRepository，并将接口标记为 @NoRepositoryBean，以防被当作一般的 Repository 调用，如代码清单 4-5 所示。接口 ExpandJpaRepository 不仅扩展了 JPA 原来的 findOne、findAll、count 等方法，而且增加了 deleteByIds、getEntityClass、nativeQuery4Map 等方法，其中 nativeQuery4Map 用来执行原生的复杂的 SQL 查询语句。

代码清单 4-5　扩展 JPA 接口定义

```
@NoRepositoryBean
public interface ExpandJpaRepository<T, ID extends Serializable> extends
JpaRepository<T,ID> {
    T findOne(String condition, Object... objects);

    List<T> findAll(String condition, Object... objects);

    List<T> findAll(Iterable<Predicate> predicates, Operator operator);

     List<T> findAll(Iterable<Predicate> predicates, Operator operator, Sort
sort);

    Page<T> findAll(Iterable<Predicate> predicates, Operator operator, Pageable
pageable);

    long count(Iterable<Predicate> predicates, Operator operator);

    List<T> findAll(String condition, Sort sort, Object... objects);

    Page<T> findAll(String condition, Pageable pageable, Object... objects);

    long count(String condition, Object... objects);

    void deleteByIds(Iterable<ID> ids);

    Class<T> getEntityClass();

    List<Map<String,Object>> nativeQuery4Map(String sql);

    Page<Map> nativeQuery4Map(String sql, Pageable pageable);

Object nativeQuery4Object(String sql);
}
```

这一接口的所有声明方法，必须由我们来实现。为了节省篇幅，只列出实现的部分代码，如代码清单 4-6 所示。完整的代码可以通过检出实例工程查看。实现代码中使用了 JPQL 查询语言（Java Persistence Query Language），它是 JPA 的查询语句规范。

代码清单 4-6 扩展 JPA 接口实现

```
    public class ExpandJpaRepositoryImpl<T,ID extends Serializable> extends
SimpleJpaRepository<T,ID> implements ExpandJpaRepository<T,ID> {
        private final EntityManager entityManager;
        private final JpaEntityInformation<T, ?> entityInformation;

        public ExpandJpaRepositoryImpl(JpaEntityInformation<T, ?> entityInformation,
EntityManager entityManager) {
            super(entityInformation, entityManager);
            this.entityManager = entityManager;
            this.entityInformation = entityInformation;
        }

        @Override
        public T findOne(String condition, Object... values) {
            if(isEmpty(condition)){
                throw new NullPointerException("条件不能为空！");
            }
            T result = null;
            try {
                result = (T) createQuery(condition, values).getSingleResult();
            } catch (NoResultException e) {
                e.printStackTrace();
            }
            return result;
        }

        @Override
        public List<T> findAll(Iterable<Predicate> predicates, Operator operator) {
            return new JpqlQueryHolder(predicates,operator).createQuery().getResult
List();
        }

        @Override
        public List<T> findAll(Iterable<Predicate> predicates, Operator operator,
Sort sort) {
            return new JpqlQueryHolder(predicates,operator,sort).createQuery().getResult
List();
        }
```

```
    @Override
    public Page<T> findAll(Iterable<Predicate> predicates, Operator operator, Pageable
pageable) {
        if(pageable==null){
            return new PageImpl<T>((List<T>) findAll(predicates,operator));
        }

        Long total = count(predicates,operator);

        Query query = new JpqlQueryHolder(predicates,operator,pageable.getSort()).
createQuery();
        query.setFirstResult(pageable.getOffset());
        query.setMaxResults(pageable.getPageSize());

        List<T> content = total > pageable.getOffset() ? query.getResultList() :
Collections.<T> emptyList();

        return new PageImpl<T>(content, pageable, total);
    }
    ......
}
```

因为自定义的接口继承于 JpaRepository，所以不但具有自定义的一些功能，而且拥有 JPA 原来的所有功能，它的继承关系如图 4-3 所示。

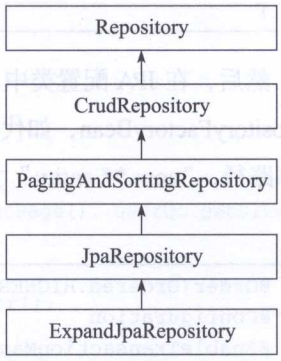


图 4-3　扩展 JPA 接口的继承关系

4.2.2　装配自定义的扩展接口

自定义的接口必须在程序启动时装配，才能正常使用。首先，创建一个装配类 ExpandJpaRepository-FactoryBean，继承于 JpaRepositoryFactory-Bean，用来加载自定义的扩展接口，如代码清单 4-7 所示。其中 getTargetRepository 返回自定义的接口实现：ExpandJpaRepositoryImpl。

代码清单 4-7　JPA 扩展接口装配类

```
public class ExpandJpaRepositoryFactoryBean<R extends JpaRepository<T,
ID>, T, ID extends Serializable>
        extends JpaRepositoryFactoryBean<R, T, ID> {

    protected RepositoryFactorySupport createRepositoryFactory(
            EntityManager entityManager) {
        return new ExpandJpaRepositoryFactory<T, ID>(entityManager);
```

```
    }
    private static class ExpandJpaRepositoryFactory<T, ID extends Serializable>
            extends JpaRepositoryFactory {

        private final EntityManager entityManager;

        public ExpandJpaRepositoryFactory(EntityManager entityManager) {

            super(entityManager);
            this.entityManager = entityManager;
        }

        protected Object getTargetRepository(RepositoryMetadata metadata) {
            JpaEntityInformation<T, Serializable> entityInformation = (JpaEntity
Information<T, Serializable>) getEntityInformation(metadata.getDomainType());
            return new ExpandJpaRepositoryImpl<T, ID>(entityInformation, entity
Manager);
        }

        protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {
            return ExpandJpaRepositoryImpl.class;
        }
    }
}
```

然后，在 JPA 配置类中，通过 @EnableJpaRepositories 加载定义的装配类 ExpandJpaRepositoryFactoryBean，如代码清单 4-8 所示。其中，“com.**.repository”为定义接口的资源库路径，“com.**.entity”为实体模型的路径。

代码清单 4-8 JPA 配置类

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@EnableTransactionManagement(proxyTargetClass = true)
@EnableJpaRepositories(basePackages = "com.**.repository",repositoryFactor
yBeanClass = ExpandJpaRepositoryFactoryBean.class)
@EntityScan(basePackages = "com.**.entity")
public class JpaConfiguration {
    @Bean
    PersistenceExceptionTranslationPostProcessor persistenceExceptionTrans
lationPostProcessor(){
        return new PersistenceExceptionTranslationPostProcessor();
    }
}
```

4.2.3　使用扩展接口

现在来做实体的持久化，这样就可以直接使用自定义的扩展接口了。如代码清单 4-9 所示，资源库接口 UserRepository 继承的就是前面定义的接口 ExpandJpaRepository。

代码清单 4-9　使用扩展接口做持久化

```
@Repository
public interface UserRepository extends ExpandJpaRepository<User, Long> {
    @Query("select t from User t where t.name =?1 and t.email =?2")
    User findByNameAndEmail(String name, String email);

    @Query("select t from User t where t.name like :name")
    Page<User> findByName(@Param("name") String name, Pageable pageRequest);
}
```

使用 JPA 扩展接口与使用原来的 JPA 接口一样，调用方法基本相同，只不过有些方法被赋予更为丰富的功能，可以更加灵活地使用。代码清单 4-10 是一个使用扩展接口的分页查询，使用 PredicateBuilder 来构造一个查询参数的对象，它可以包含更多的查询参数。

代码清单 4-10　使用扩展 JPA 接口的分页查询

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public Page<User> findPage(UserQo userQo){
        Pageable pageable = new PageRequest(userQo.getPage(), userQo.getSize(),
new Sort(Sort.Direction.ASC, "id"));

        PredicateBuilder pb  = new PredicateBuilder();

        if (!StringUtils.isEmpty(userQo.getName())) {
            pb.add("name","%" + userQo.getName() + "%", LinkEnum.LIKE);
        }
        if (!StringUtils.isEmpty(userQo.getCreatedateStart())) {
            pb.add("createdate",userQo.getCreatedateStart(), LinkEnum.GE);
        }
        if (!StringUtils.isEmpty(userQo.getCreatedateEnd())) {
            pb.add("createdate",userQo.getCreatedateEnd(), LinkEnum.LE);
        }

        return userRepository.findAll(pb.build(), Operator.AND, pageable);
```

```
    }
}
```

4.3 使用 Redis 做缓存

在数据库使用中，数据查询是最大的性能开销。如果能借助 Redis 作为辅助缓存，将可以极大地提高数据库的访问性能。使用 Redis 做缓存，一方面可以像第 2 章介绍的使用 Redis 那样调用，另一方面，可以使用注解的方式来调用，这种方式更加简单，代码也更加简洁。

需要注意的是，Redis 是一个具有持久化功能的数据库系统，若使用默认配置，存取的数据就会永久地保存在磁盘中。如果只是使用 Redis 来做缓存，并不需要 Redis 永久保存数据，可以设定在 Redis 保存数据的期限来实现，这样，过期的数据将自动从 Redis 数据库中清除。这不但能很好地利用 Redis 的快速存取功能，而且能彻底减轻 Redis 的负担。作为缓存使用的数据，最初就是从数据库中查询出来的，所以完全没有必要再做一次永久保存。始终让 Redis 保持轻装上阵，才能最好地发挥它的性能优势。

4.3.1 使用 Spring Cache 注解

结构简单的对象，即没有包含其他对象的实体，可以使用 Spring Cache 注解的方式来使用 Redis 缓存。要使用注解的方式调用缓存，必须在配置类中启用 Spring Cache，如代码清单 4-11 所示。其中 setDefaultExpiration 指定了数据在 Redis 数据库中的有效期限。

代码清单 4-11 Spring Cache 配置

```
@Configuration
@EnableCaching
public class RedisConfig extends CachingConfigurerSupport {
        @Bean
    public CacheManager cacheManager(@SuppressWarnings("rawtypes") RedisTemplate
redisTemplate) {
            RedisCacheManager manager = new RedisCacheManager(redisTemplate);
            manager.setDefaultExpiration(43200);//12 小时
            return manager;
    }
}
```

这样，就可以在对数据接口的调用中，对增删查改加入如代码清单 4-12 所示的注

解，自动增加缓存的创建、修改和删除等功能。其中注解 @Cacheable 为存取缓存，注解 @CachePut 为修改缓存，如果不存在则创建，注解 @CacheEvict 为删除缓存，当删除数据时，如果缓存还存在，就必须删除，各个注解中的 value 参数是一个 key 的前缀，并由 keyGenerator 按一定的规则生成一个唯一标识。

代码清单 4-12　用注解方式使用 Redis 做缓存

```
@Service
public class RoleService {
    @Autowired
    private RoleRepository roleRepository;
    @Autowired
    private RoleRedis roleRedis;

    @Cacheable(value = "mysql:findById:role", keyGenerator = "simpleKey")
    public Role findById(Long id) {
        return roleRepository.findOne(id);
    }

    @CachePut(value = "mysql:findById:role", keyGenerator = "objectId")
    public Role create(Role role) {
        return roleRepository.save(role);
    }

    @CachePut(value = "mysql:findById:role", keyGenerator = "objectId")
    public Role update(Role role) {
        return roleRepository.save(role);
    }

    @CacheEvict(value = "mysql:findById:role", keyGenerator = "simpleKey")
    public void delete(Long id) {
        roleRepository.delete(id);
    }
    ......
}
```

对于 key 的生成规则，使用如代码清单 4-13 所示的方法来实现，这里主要使用了调用者本身对象的 ID 属性来保证它的唯一性，其中 simpleKey 和 objectId 都是提取调用者本身的类名字和参数 id 作为唯一标识。

代码清单 4-13　生成 cache 的 key

```
@Bean
    public KeyGenerator simpleKey(){
```

```
        return new KeyGenerator() {
            @Override
            public Object generate(Object target, Method method, Object... params) {
                StringBuilder sb = new StringBuilder();
                sb.append(target.getClass().getName()+":");
                for (Object obj : params) {
                    sb.append(obj.toString());
                }
                return sb.toString();
            }
        };
    }

    @Bean
    public KeyGenerator objectId(){
        return new KeyGenerator() {
            @Override
            public Object generate(Object target, Method method, Object... params){
                StringBuilder sb = new StringBuilder();
                sb.append(target.getClass().getName()+":");
                try {
                    sb.append(params[0].getClass().getMethod("getId", null).
invoke(params[0], null).toString());
                }catch (NoSuchMethodException no){
                    no.printStackTrace();
                }catch(IllegalAccessException il){
                    il.printStackTrace();
                }catch(InvocationTargetException iv){
                    iv.printStackTrace();
                }
                return sb.toString();
            }
        };
    }
```

4.3.2 使用 RedisTemplate

由于使用 Spring Cache 注解的方法使用 Redis 缓存，只能对简单对象进行系列化操作，所以对于像实体 User 这样的包含了一定关系的复杂对象，或其他集合、列表对象等，就不能使用简单注解的方法来实现了，还要像第 2 章中介绍的方法那样使用 RedisTemplate 来调用 Redis，其使用的效果和上面使用 Cache 注解的效果相同，只不过实现方法完全不同。

代码清单 4-14 使用 RedisTemplate 实现了对 Redis 的调用。这种方式可以很方便地对列表对象进行系列化，在数据存取时使用 Json 进行格式转换。这里使用分钟作为时间单位来设定数据在 Redis 中保存的有效期限。

代码清单 4-14　使用 RedisTemplate

```
@Repository
public class UserRedis {
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public void add(String key, Long time, User user) {
        Gson gson = new Gson();
        redisTemplate.opsForValue().set(key, gson.toJson(user), time, TimeUnit.
MINUTES);
    }

    public void add(String key, Long time, List<User> users) {
        Gson gson = new Gson();
        redisTemplate.opsForValue().set(key, gson.toJson(users), time, TimeUnit.
MINUTES);
    }

    public User get(String key) {
        Gson gson = new Gson();
        User user = null;
        String json = redisTemplate.opsForValue().get(key);
        if(!StringUtils.isEmpty(json))
            user = gson.fromJson(json, User.class);
        return user;
    }

    public List<User> getList(String key) {
        Gson gson = new Gson();
        List<User> ts = null;
        String listJson = redisTemplate.opsForValue().get(key);
        if(!StringUtils.isEmpty(listJson))
            ts = gson.fromJson(listJson, new TypeToken<List<User>>(){}.getType());
        return ts;
    }

    public void delete(String key){
        redisTemplate.opsForValue().getOperations().delete(key);
```

```
    }
}
```

然后编写如代码清单 4-15 所示的代码来使用 Redis 缓存。即在使用原来数据库的增删查改过程中，同时使用 Redis 进行辅助存取，以达到提升访问速度的目的，从而缓解对原来数据库的访问压力。这样，访问一条数据时，首先从 Redis 读取，如果存在则不再到 MySQL 中读取，如果不存在再到 MySQL 读取，并将读取的结果暂时保存在 Redis 中。

代码清单 4-15 在数据服务中使用 Redis 作为辅助缓存

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private UserRedis userRedis;
    private static final String keyHead = "mysql:get:user:";

    public User findById(Long id) {
        User user = userRedis.get(keyHead + id);
        if(user == null){
            user = userRepository.findOne(id);
            if(user != null)
                userRedis.add(keyHead + id, 30L, user);
        }
        return user;
    }

    public User create(User user) {
        User newUser = userRepository.save(user);
        if(newUser != null)
            userRedis.add(keyHead + newUser.getId(), 30L, newUser);
        return newUser;
    }

    public User update(User user) {
        if(user != null) {
            userRedis.delete(keyHead + user.getId());
            userRedis.add(keyHead + user.getId(), 30L, user);
        }
        return userRepository.save(user);
    }
```

```
    public void delete(Long id) {
        userRedis.delete(keyHead + id);
        userRepository.delete(id);
    }
```

上面使用 Redis 缓存的两种方法，可以在一个应用中混合使用。但不管怎么使用，对于控制器来说都是完全透明的，控制器对数据接口的调用还是像以前一样，它并不清楚数据接口后端是否启用了缓存，如代码清单 4-16 所示。

代码清单 4-16　控制器使用数据接口

```
@Autowired
    private UserService userService;

        @RequestMapping(value="/{id}")
    public String show(ModelMap model,@PathVariable Long id) {
        User user = userService.findById(id);
        model.addAttribute("user",user);
        return "user/show";
    }
```

使用缓存之后，大量的查询语句就从原来的数据库服务器中，转移到了高效的 Redis 服务器中执行，这将在很大程度上减轻原来数据库服务器的压力，并且提升查询的反应速度和效率。所以在很大的程度上，系统性能就得到了很好的改善。

4.4　Web 应用模块

对于上面一些功能的实现，最后使用一个 Web 应用来调用，以验证使用 Druid 连接池和使用 Redis 缓存的效果，同时可以体验使用 JPA 扩展接口更为丰富的功能。

4.4.1　引用数据管理模块

实例工程中的 Web 应用模块将引用数据管理模块，而数据管理模块使用了第 2 章实例工程中 MySQL 模块的实体 – 关系模型设计，即使用部门、用户和角色三个实体，如图 4-4 所示。实体的建模还与第 2 章中使用的方法一样，没有做任何修改。至于实体的持久化，如前所述，只要在原来的持久化中改变资源库接口定义中继承于自定义的扩展接口即可。

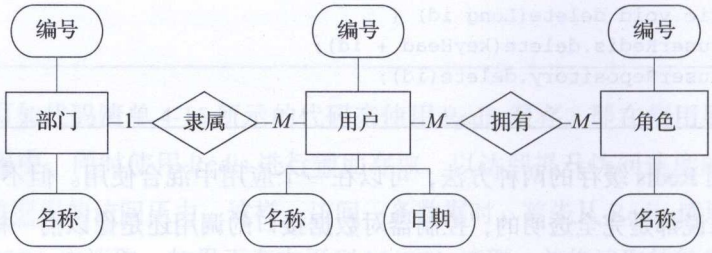


图 4-4 实体 - 关系模型设计

4.4.2 Web 应用配置

Web 应用的界面设计使用第 3 章的设计来实现。这里，主要实现对部门、用户和角色三个实体的数据进行增删查改的管理。

在 Web 应用模块的配置文件 application.yml 中，配置连接 MySQL 和 Redis 服务器的一些参数，如代码清单 4-17 所示。

代码清单 4-17 Web 应用配置

```
server:
    port: 80
    tomcat:
        uri-encoding: UTF-8

spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
        url: jdbc:mysql:// localhost:3306/test?characterEncoding=utf8
        username: root
        password: 12345678
        # 初始化大小，最小，最大
        initialSize: 5
        minIdle: 5
        maxActive: 20
        # 配置获取连接等待超时的时间
        maxWait: 60000
        # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
        timeBetweenEvictionRunsMillis: 60000
        # 配置一个连接在池中的最小生存时间，单位是毫秒
        minEvictableIdleTimeMillis: 300000
        validationQuery: SELECT 1 FROM DUAL
        testWhileIdle: true
```

```
    testOnBorrow: false
    testOnReturn: false
    # 打开 PSCache，并且指定每个连接上 PSCache 的大小
    poolPreparedStatements: true
    maxPoolPreparedStatementPerConnectionSize: 20
    # 配置监控统计拦截的 filters，去掉后监控界面 SQl 将无法统计，'wall' 用于防火墙
    filters: stat,wall,log4j
    # 通过 connectProperties 属性来打开 mergeSql 功能；慢 SQL 记录
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
    # 合并多个 DruidDataSource 的监控数据
    #useGlobalDataSourceStat=true

jpa:
    database: MYSQL
    show-sql: true
## Hibernate ddl auto (validate|create|create-drop|update)
    hibernate:
        ddl-auto: update
        naming-strategy: org.hibernate.cfg.ImprovedNamingStrategy
    properties:
        hibernate:
        dialect: org.hibernate.dialect.MySQL5Dialect
## redis
redis:
    host: 192.168.1.214
    port: 6379
    pool:
        max-idle: 8
        min-idle: 0
        max-active: 8
        max-wait: -1
```

启动应用后，运行效果如图 4-5 所示。除了分页数据没有做缓存之外，其他查询都做了缓存处理。在控制台上可以看到执行的 SQL 查询语句，一个查询，比如查看用户，如果在控制台上没有看到输出查询语句，就可以说明是调用了 Redis 缓存。

关于使用缓存的情况，也可以登录安装 Redis 的服务器，使用下列指令，查看当前所有的 key。

```
#redis-cli
>keys *
```

下载一个 Redis 客户端，可以更加直观地查看 Redis 服务器的情况，如图 4-6 所示。

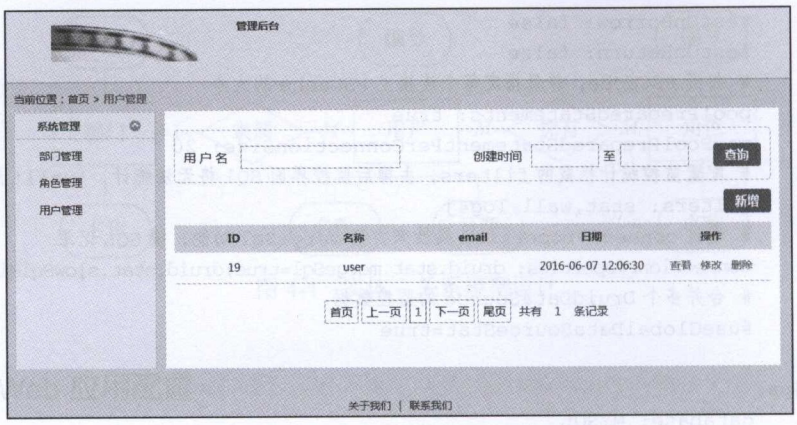


图 4-5 Web 应用运行效果

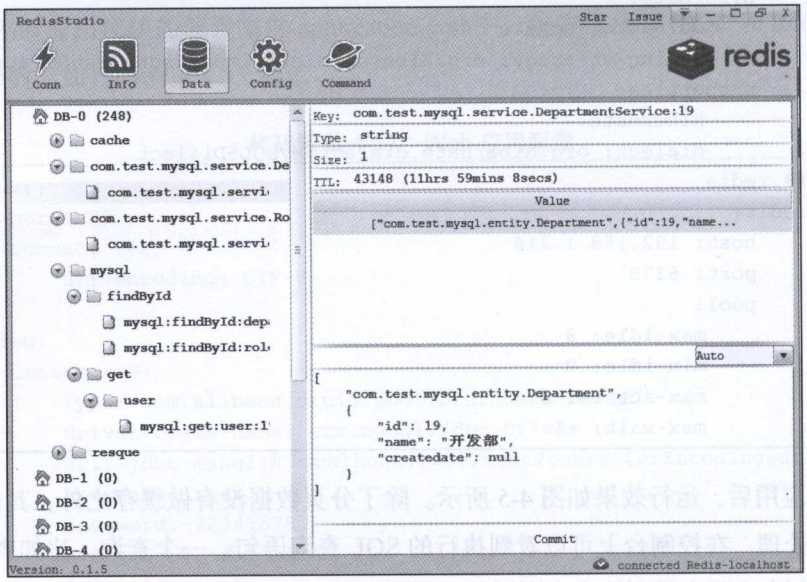


图 4-6 Redis 客户端

4.5 运行与发布

本章实例的完整代码可以直接在 IDEA 中通过 GitHub 检出：https://github.com/chen-fromsz/spring-boot-dbup.git。

检出工程后，可以运行 Web 应用模块 website 进行测试。在 IDEA 的 Run/Configura-

tion 中增加一个 Spring Boot 配置，模块选择 website，工作目录选择 website 模块所在的路径，主程序选择 com.test.website.WebApplication，并将配置保存为 webapp。

在 MySQL 服务器中创建一个数据库：test，配置 Web 应用模块 website 的配置文件 application.yml 中连接 MySQL 服务器的 url、username、password，以及 Redis 的 host 和 port。然后运行配置项目 webapp，启动完成后，在浏览器中打开网址：http://127.0.0.1

注意　因为 localhost 不能加入 Druid 的监控服务器的白名单中，所以使用 localhost 可能不能正常访问。而使用域名的方式是可以的，只要把域名所指向的 IP 加入 Druid 的白名单中即可。

如果要打包发布，可以在 IDEA 的 Run/Configuration 中增加一个 Maven 配置项目，工作目录选择工程根目录 spring-boot-dbup 所在的路径，在命令行中输入指令 clean package，然后将配置项目保存为 mvn。或者直接打开一个命令行窗口，切换到工程根目录所在路径，执行下列 Maven 指令：

```
mvn clean package
```

打包完成后，在命令行窗口中切换到模块 website 的 target 目录中，输入下列指令可运行应用：

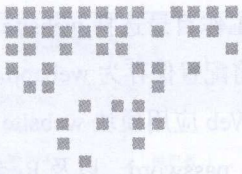
```
java -jar website-1.0-SNAPSHOT.jar
```

4.6　小结

本章使用 Druid 连接池和 Redis 数据库作为缓存，提升了关系型数据库的访问性能，并且通过扩展全局的 JPA 接口，丰富了资源库的调用功能。

对于大数据时代的互联网应用来说，要从根本上提升数据库的性能，主要还在于数据库本身的设计和配置上，例如使用分布式设计的集群系统，通过合理的配置和组装，可以达到横向扩展的目的，以后通过增加设备的方式，可以随时扩充数据库的容量和提高其访问性能。

有了完备的数据库访问功能和漂亮的操作界面之后，一个应用中更重要的设计就是安全设计了。下一章将介绍使用 Spring Security 来为一个应用实现安全设计，从而实现用户认证和权限管理方面的功能。



Chapter 5 第 5 章

Spring Boot 安全设计

Web 应用的安全管理，主要包括两个方面的内容：一方面是用户身份认证，即用户登录的设计；另一方面是用户授权，即一个用户在一个应用系统中能够执行哪些操作的权限管理。权限管理的设计一般使用角色来管理，即给一个用户赋予哪些角色，这个用户就具有哪些权限。本章主要使用 spring-cloud-security 来进行安全管理设计。下面首先了解安全设计的依赖配置管理。

5.1 依赖配置管理

为了更方便地使用 spring-cloud-security，将使用 Spring Cloud（关于 Spring Cloud 将在后面的章节中介绍）的 Maven 依赖配置，如代码清单 5-1 所示。Spring Cloud 有两个版本，第一版本的代号是 Angel，第二个版本的代号是 Brixton，这两个版本又包含各自的子版本，将使用 Brixton.M5，因为它包含了 Spring Boot 1.3.2，这和前面章节使用 Spring Boot 的版本相同，同时 Spring Security 默认的版本是 4.0 以上。

代码清单 5-1　Maven 依赖配置

```
<parent>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-parent</artifactId>
        <version>Brixton.M5</version>
```

```
        <relativePath/>
</parent>
.....
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

了解依赖配置管理后，学习如何配置安全策略。

5.2　安全策略配置

关于系统的安全管理及各种设计，Spring Security 已经大体上都实现了，只需要进行一些配置和引用，就能够正常使用。如代码清单 5-2 所示，安全配置类 SecurityConfiguration 继承了 Spring Security 的 WebSecurityConfigurerAdapter。这里可以使用 HttpSecurity 的一些安全策略进行配置，各项配置的解释如下：

- loginPage：设置一个使用自定义的登录页面 URL。
- loginSuccessHandler：设置自定义的一个登录成功处理器。
- permitAll：是完全允许访问的一些 URL 配置，并可以使用通配符来设置，这里将一些资源目录赋予可以完全访问的权限，由 settings 指定的权限列表也赋予了完全访问的权限。
- logout：设置使用默认的登出。
- logoutSuccessUrl：设定登出成功的链接。
- rememberMe：用来记住用户的登录状态，即用户没有执行退出时，再次打开页面将不用登录。
- csrf：即跨站请求伪造（cross-site request forgery），这是一个防止跨站请求伪造攻击的策略设置。
- accessDeniedPage：配置一个拒绝访问的提示链接。

其中，settings 是引用了自定义的配置参数。

代码清单 5-2　安全策略配置

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    private SecuritySettings settings;
```

```
        @Override
        protected void configure(HttpSecurity http) throws Exception {
            http.formLogin().loginPage("/login").permitAll().successHandler(login
SuccessHandler())
                    .and().authorizeRequests()
                    .antMatchers("/images/**", "/checkcode", "/scripts/**", "/
styles/**").permitAll()
                    .antMatchers(settings.getPermitall().split(",")).permitAll()
                    .anyRequest().authenticated()
                    .and().csrf().requireCsrfProtectionMatcher(csrfSecurityReq
uestMatcher())
                    .and().sessionManagement().sessionCreationPolicy(SessionCr
eationPolicy.NEVER)
                    .and().logout().logoutSuccessUrl(settings.getLogoutsuccssurl())
                    .and().exceptionHandling().accessDeniedPage(settings.getDenied
page())
                    .and().rememberMe().tokenValiditySeconds(1209600).tokenRep
ository(tokenRepository());
        }
        ......
    }
```

5.2.1 权限管理规则

代码清单 5-2 中引用的 SecuritySettings 是自定义的一个配置类，如代码清单 5-3 所示。其中使用注解 @ConfigurationProperties 设定配置参数的前缀部分为 securityconfig，定义的各个配置参数的意义如下：

- logoutsuccessurl：用来定义退出成功的链接。
- permitall：用来定义允许访问的 URL 列表。
- deniedpage：用来设定拒绝访问的信息提示链接。
- urlroles：这是一个权限管理规则，是链接地址与角色权限的配置列表。

代码清单 5-3 自定义配置类

```
@ConfigurationProperties(prefix="securityconfig")
public class SecuritySettings {
    private String logoutsuccessurl = "/logout";
    private String permitall = "/api";
    private String deniedpage = "/deny";
    private String urlroles;
    ......
}
```

使用自定义配置参数后，可以在工程的配置文件 application.yml 中对安全管理进行集中配置，如代码清单 5-4 所示。

代码清单 5-4　使用自定义的 securityconfig 配置

```
securityconfig:
    logoutsuccssurl: /
    permitall: /rest/**,/bbs**
    deniedpage: /deny
    urlroles: /**/new = manage,admin;
              /**/edit/** = admin;
              /**/delete/** = admin
```

其中 urlroles 配置一个权限配置列表，这是我们设计的一种权限管理规则，列表中的每一个配置项用分号分隔，每一个配置项的等号左边是一个可以带上通配符的链接地址，等号右边是一个角色列表，角色之间用逗号分隔。每一个配置项表示包含等号左边字符串的链接地址，能够被等号右边的角色访问。

这将要求我们的控制器设计链接地址时，必须遵循这一权限管理规则，这样只要使用一个简单的配置列表，就能够覆盖整个系统的权限管理策略。设计控制器链接地址的规则如下，它包含了系统增删查改的所有操作。

- /**/new：新建；
- /**/edit/**：修改；
- /**/delete/**：删除；
- /**/show/**：查看；
- /**/list：列表查询。

使用这种规则之后，再来看看代码清单 5-4 中 urlroles 的权限配置，这里只需要简单的三个配置项，就已经完成了对一个应用系统所有权限的管理配置了。其中，新建操作只有 manage、admin 两个角色有权限，修改操作和删除操作只有 admin 这个角色有权限，至于没有在权限管理列表中配置的查看操作，因为没有限定角色访问，所以它能被所有用户访问。

这种权限策略配置好了之后，要让应用系统中的一个用户具有哪些权限，只要分配给这个用户一些角色就可以。

5.2.2　登录成功处理器

登录成功后，如果需要对用户的行为进行记录或者执行其他操作，可以使用登录成

功处理器。代码清单 5-5 是一个登录成功处理器的定义，这里只是简单地输出了用户登录的日志。

代码清单 5-5 登录成功处理器

```
    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
HttpServletResponse response, Authentication authentication)
                throws IOException,ServletException {
            User userDetails = (User)authentication.getPrincipal();

            log.info("登录用户 user:" + userDetails.getName() + "login"+request.
getContextPath());
            log.info("IP:" + getIpAddress(request));
            super.onAuthenticationSuccess(request, response, authentication);
    }
```

5.2.3 防攻击策略

因为 Spring Security 的跨站请求伪造（cross-site request forgery，CSRF）即阻止跨站请求伪造攻击的功能很完善，所以使用 Spring Security 之后，对于新建、修改和删除等操作，必须进行特殊的处理，才能正常使用。这要求在所有具有上面操作请求的页面上提供如下代码片段，因为我们的页面设计使用了 Thymeleaf 模板，所以只要在 layout.hmtl 的页头上加入下面两行代码即可，loyou.html 是所有页面都会用到的一个页面文件。

```
<meta name="_csrf" th:content="${_csrf.token}"/>
<meta name="_csrf_header" th:content="${_csrf.headerName}"/>
```

还要在 loyout.html 中引用脚本文件 public.js，然后在 public.js 中增加一个函数，如代码清单 5-6 所示。这样做的意思是，在表单提交时放入一个 token，服务端验证该 token 是否有效，只允许有效的 token 请求，否则拒绝当前操作。这样就能够很好地起到防御 CSRF 攻击的目的。

代码清单 5-6 阻止 CSRF 攻击策略

```
$(function () {
    var token = $("meta[name='_csrf']").attr("content");
    var header = $("meta[name='_csrf_header']").attr("content");
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(header, token);
```

```
    });
});
```

如果要对第三方开放接口，上面的方法就不适用了，这时只能对特定的 URL 使用排除 CSRF 保护的方法来实现。代码清单 5-7 对指定的 URL 排除对其进行 CSRF 的保护。

代码清单 5-7　排除 CSRF 保护策略

```
public class CsrfSecurityRequestMatcher implements RequestMatcher {
    protected Log log = LogFactory.getLog(getClass());
    private Pattern allowedMethods = Pattern
            .compile("^(GET|HEAD|TRACE|OPTIONS)$");
    /**
     * 需要排除的 url 列表
     */
    private List<String> excludeUrls;

    @Override
    public boolean matches(HttpServletRequest request) {
        if (excludeUrls != null && excludeUrls.size() > 0) {
            String servletPath = request.getServletPath();
            for (String url : excludeUrls) {
                if (servletPath.contains(url)) {
                    log.info("++++"+servletPath);
                    return false;
                }
            }
        }
        return !allowedMethods.matcher(request.getMethod()).matches();
    }
        ......
}
```

然后在配置类中，加入需要排除阻止 CSRF 攻击的链接列表，如代码清单 5-8 所示，只要链接地址中包含“/rest”字符串，就将对其忽略 CSRF 保护策略。

代码清单 5-8　在安全配置类加入需要排除 CSRF 保护的列表

```
    private CsrfSecurityRequestMatcher csrfSecurityRequestMatcher(){
        CsrfSecurityRequestMatcher csrfSecurityRequestMatcher = new CsrfSecurity
RequestMatcher();
        List<String> list = new ArrayList<String>();
        list.add("/rest/");
        csrfSecurityRequestMatcher.setExcludeUrls(list);
```

```
        return csrfSecurityRequestMatcher;
    }
```

5.2.4 记住登录状态

代码清单 5-2 中的安全策略配置中有一行配置：rememberMe().tokenValiditySeconds(86400).tokenRepository(tokenRepository())，它是用来记住用户登录状态的一个配置，其中 86400 指定记住的时间秒数，即为 1 天时间。为了实现这个功能，需要将一个用户的登录令牌等信息保存在数据库中，这需要在配置类中指定连接数据库的数据源，如代码清单 5-9 所示。

代码清单 5-9 指定保存登录用户 token 的数据源

```
@Autowired @Qualifier("dataSource")
private DataSource dataSource;

@Bean
    public JdbcTokenRepositoryImpl tokenRepository(){
        JdbcTokenRepositoryImpl jtr = new JdbcTokenRepositoryImpl();
        jtr.setDataSource(dataSource);
        return jtr;
    }
```

同时，还应该在数据库中增加一个数据表 persistent_logins，这个表结构的定义是由 Spring Security 提供的，使用一个实体来实现，这样做的目的只是为了在系统启动时能够创建这个表结构而已，如代码清单 5-10 所示，它用来保存用户名、令牌和最后登录时间等信息。

代码清单 5-10 记住用户登录状态的实体建模

```
@Entity
@Table(name = "persistent_logins")
public class PersistentLogins implements java.io.Serializable{
    @Id
    @Column(name = "series", length = 64, nullable = false)
    private String series;
    @Column(name = "username", length = 64, nullable = false)
    private String username;
    @Column(name = "token", length = 64, nullable = false)
    private String token;
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "last_used", nullable = false)
```

```
    private Date last_used;
    ......
    }
```

5.3　登录认证设计

完成上面的安全策略配置之后，打开受保护的页面或链接时，就会引导用户到登录页面上输入用户名和密码验证用户身份。如果在安全配置中没有指定登录页面 URL，Spring Security 就调用其默认的登录页面。只是，Spring Security 的登录页面设计很简单，不适合于一般的 Web 应用的登录设计。除了登录页面，Spring Security 对于用户身份验证同样也已经实现了，只需要加以引用即可。

5.3.1　用户实体建模

可以使用第 2 章的实例工程 MySQL 模块的实体建模来建立用户体系，回顾一下，在第 2 章中建模的实体中包含用户、部门和角色三个对象，它们的关系是，一个用户只能属于一个部门，一个用户可以拥有多个角色，这非常适合本章的实例。除了部门和角色，用户实体的属性必须做些调整，以适合本章实例的要求，如代码清单 5-11 所示，即增加了邮箱、性别和密码等几个属性，其他基本相同。

代码清单 5-11　用户实体建模

```
@Entity
@Table(name = "user")
public class User implements java.io.Serializable{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    private Integer sex;
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date createdate;
    private String password;

    @ManyToOne
    @JoinColumn(name = "did")
    @JsonBackReference
```

```
    private Department department;

    @ManyToMany(cascade = {}, fetch = FetchType.EAGER)
    @JoinTable(name = "user_role",
            joinColumns = {@JoinColumn(name = "user_id")},
            inverseJoinColumns = {@JoinColumn(name = "roles_id")})
    private List<Role> roles;
    ......
}
```

另外，在用户实体的持久化方面，也增加了几个方法以便能适用本章实例的要求，如代码清单 5-12 所示。其中 User findByName（String name）就是登录时使用用户名来查询用户的信息。

代码清单 5-12 用户实体持久化接口

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    @Query("select t from User t where t.name =?1 and t.email =?2")
    User findByNameAndEmail(String name, String email);

    @Query("select t from User t where t.name like :name")
    Page<User> findByName(@Param("name") String name, Pageable pageRequest);

    User findByName(String name);
}
```

5.3.2 用户身份验证

在安全配置类的定义中，使用了如代码清单 5-13 所示的配置，用来调用我们自定义的用户认证 CustomUserDetailsService，并且指定了使用密码的加密算法为 BCryptPasswordEncoder，这是 Spring Security 官方推荐的加密算法，比 MD5 算法的安全性更高。

代码清单 5-13 安全配置类引用 CustomUserDetailsService

```
@Autowired
private CustomUserDetailsService customUserDetailsService;

@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
    auth.userDetailsService(customUserDetailsService).passwordEncoder
```

```
(passwordEncoder());
            // remember me
            auth.eraseCredentials(false);
    }

        @Bean
        public BCryptPasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }
```

如代码清单 5-14 所示，CustomUserDetailsService 实现了 Spring Security 的 UserDetailsService，重载了 loadUserByUsername(String userName)，并返回自定义的 SecurityUser，通过这个 SecurityUser 来完成用户的身份认证。其中，loadUserByUsername 调用了用户资源库接口的 findByName 方法，取得登录用户的详细信息。

代码清单 5-14　CustomUserDetailsService 定义

```
    @Component
    public class CustomUserDetailsService implements UserDetailsService {
        @Autowired
        private UserRepository userRepository;

        @Override
        public UserDetails loadUserByUsername(String userName) throws UsernameNot
FoundException {
            User user = userRepository.findByName(userName);
            if (user == null) {
                throw new UsernameNotFoundException("UserName " + userName + "
not found");
            }
            return new SecurityUser(user);
        }
    }
```

SecurityUser 继承于实体对象 User，并实现了 Spring Security 的 UserDetails，同时重载了 getAuthorities()，用来取得为用户分配的角色列表，用于后面的权限验证，它的实现如代码清单 5-15 所示。

代码清单 5-15　SecurityUser 定义

```
    public class SecurityUser extends User implements UserDetails
    {
        @Override
        public Collection<? extends GrantedAuthority> getAuthorities() {
```

```
            Collection<GrantedAuthority> authorities = new ArrayList<Granted
Authority>();
            List<Role> roles = this.getRoles();
            if(roles != null)
            {
                for (Role role : roles) {
                    SimpleGrantedAuthority authority = new SimpleGrantedAuthority
(role.getName());
                    authorities.add(authority);
                }
            }
            return authorities;
        }
        ......
    }
```

5.3.3 登录界面设计

首先创建一个登录控制器，编写如代码清单 5-16 所示的代码，这个控制器很简单，它仅仅是返回对一个页面的调用，页面的设计文件是 login.html。

代码清单 5-16 登录控制器

```
@Controller
public class LoginController {
    @RequestMapping("/login")
    public String login(){
        return "login";
    }
}
```

登录界面的设计在页面文件 login.html 中完成，代码清单 5-17 是表单设计的部分代码和一些错误提示设计。表单中设置了用户、密码和验证码等输入框，最终使用 POST 方式提交，提交的链接地址是 /loging，这将请求 Spring Security 的内部方法。

代码清单 5-17 登录界面表单设计

```
                <div th:if="${param.error}">
                <input th:value=" 无效的用户或密码！ " id="errorMsg" type="hidden"/>
            </div>
            <div th:if="${param.logout}">
                <input th:value=" 你已经退出！ " id="errorMsg" type="hidden"/>
            </div>
            <div th:if="${#httpServletRequest.remoteUser != null}">
```

```
                <input th:value="${#httpServletRequest.remoteUser}" id="errorMsg"
type="hidden"/>
            </div>
    <form th:action="@{/login}" id="loginForm" method="post">
                <div class="loginTit png"></div>
                <ul class="infList">
                    <li class="grayBox">
                        <label for="username" class="username-icon"></label>
                        <input id="username" class="username" name="username"
type="text" placeholder=" 您的用户名 "/>
                        <div class="close png hide"></div>
                    </li>
                    <li class="grayBox">
                        <label class="pwd-icon" id="pwd"></label>
                        <input id="password" name="password" class="pwd" type=
"password" placeholder=" 登录密码 "/>
                        <div class="close png hide"></div>
                    </li>
                    <li class="">
                        <label class="validateLabel" ></label>
                        <input id="checkCode" name="checkCode" class="checkCode"
type="text" placeholder=" 验证码 " />
                        <img onclick="reloadImg();" style="cursor: pointer"
th:src="@{/images/imagecode}" id="validateImg" alt=" 验证码 " class="codePic" title=
" 验证码。单击此处更新验证码。"/>
                        <a class="getOther" href="javascript:void(0);" onclick=
"reloadImg();" title=" 单击此处可以更新验证码。"> 更新 </a>
                    </li>
                </ul>
                <ul class="infList reloadBtn" style="display: none;">
                    <li>
                        <a href="javascript:void(0);" onclick="tologin();">
本页面已经失效。请单击此处重新登录。</a>
                    </li>
                </ul>
                <div class="loginBtnBox">
                    <div class="check-box"><input type="hidden" value="0" id=
"remember-me" name="remember-me" onclick="if(this.checked){this.value = 1}
else{this.value=0}" /><span class="toggleCheck no-check" id="repwd"></span> 记住
我 </div>
                    <input type="button" id="loginBtn" onclick="verSubmit()"
value=" 登录 " class="loginBtn png" />
                </div>
            </form>
```

完成的登录界面设计效果如图 5-1 所示。

5.3.4 验证码验证

注意到上面登录界面设计中有一个验证码功能，这个功能 Spring Security 是没有的，必须由我们来实现。代码清单 5-18 是使用验证码的实现代码，其中 imagecode 方法是一个生成图形验证码的请求，checkcode 方法实现了对这个图形验证码的验证。从验证码的生成到验证的过程中，验证码是通过 Session 来保存的，并且设定一个验证码的最长有效时间为 5 分钟。验证码的生成规则是从 0 ~ 9 的数字中，随机产生一个 4 位数，并增加一些干扰元素，最终组合成为一个图形输出。

图 5-1 登录界面设计效果图

代码清单 5-18 验证码验证

```
    @RequestMapping(value = "/images/imagecode")
        public String imagecode(HttpServletRequest request, HttpServletResponse
response)
                throws Exception {
            OutputStream os = response.getOutputStream();
            Map<String,Object> map = ImageCode.getImageCode(60, 20, os);

            String simpleCaptcha = "simpleCaptcha";
            request.getSession().setAttribute(simpleCaptcha, map.get("strEnsure").
toString().toLowerCase());
            request.getSession().setAttribute("codeTime",new Date().getTime());

            try {
                ImageIO.write((BufferedImage) map.get("image"), "JPEG", os);
            } catch (IOException e) {
                return "";
            }
            return null;
        }

        @RequestMapping(value = "/checkcode")
        @ResponseBody
        public String checkcode(HttpServletRequest request, HttpSession session)
                throws Exception {
            String checkCode = request.getParameter("checkCode");
            Object cko = session.getAttribute("simpleCaptcha") ; //验证码对象
            if(cko == null){
```

```
            request.setAttribute("errorMsg", "验证码已失效，请重新输入！");
            return "验证码已失效，请重新输入！";
        }

        String captcha = cko.toString();
        Date now = new Date();
        Long codeTime = Long.valueOf(session.getAttribute("codeTime")+"");
        if(StringUtils.isEmpty(checkCode) || captcha == null ||  !(checkCode.
equalsIgnoreCase(captcha))){
            request.setAttribute("errorMsg", "验证码错误！");
            return "验证码错误！";
        }else if ((now.getTime()-codeTime)/1000/60>5){// 验证码有效时长为 5 分钟
            request.setAttribute("errorMsg", "验证码已失效，请重新输入！");
            return "验证码已失效，请重新输入！";
        }else {
            session.removeAttribute("simpleCaptcha");
            return "1";
        }
    }
```

5.4　权限管理设计

用户通过身份认证，成功登录系统后，就要开始检查用户访问资源的权限，如果用户没有权限访问，将会阻止用户访问受保护的资源，并给出错误提示信息。

5.4.1　权限管理配置

在安全配置类中，定义了几个类，实现自定义的权限检查判断及其管理的功能，如代码清单 5-19 所示，各个类的意义如下：

- CustomFilterSecurityInterceptor：权限管理过滤器。
- CustomAccessDecisionManager：权限管理决断器。
- CustomSecurityMetadataSource：权限配置资源管理器。

其中，过滤器在系统启动时开始工作，并同时导入资源管理器和权限决断器，对用户访问的资源进行管理。权限决断器对用户访问的资源与用户拥有的角色权限进行对比，以此来判断一个用户是否对一个资源具有访问权限。

代码清单 5-19　安全配置类中的权限管理设置

```
    @Bean
        public CustomFilterSecurityInterceptor customFilter() throws Exception{
```

```
        CustomFilterSecurityInterceptor customFilter = new CustomFilterSec
urityInterceptor();
        customFilter.setSecurityMetadataSource(securityMetadataSource());
        customFilter.setAccessDecisionManager(accessDecisionManager());
        customFilter.setAuthenticationManager(authenticationManager);
        return customFilter;
    }

    @Bean
    public CustomAccessDecisionManager accessDecisionManager() {
        return new CustomAccessDecisionManager();
    }

    @Bean
    public CustomSecurityMetadataSource securityMetadataSource() {
        return new CustomSecurityMetadataSource(settings.getUrlroles());
    }
```

5.4.2 权限管理过滤器

权限管理过滤器继承于 Spring Security 的 AbstractSecurityInterceptor，实时监控用户的行为，防止用户访问未被授权的资源，如代码清单 5-20 所示。

代码清单 5-20 权限管理过滤器

```
public class CustomFilterSecurityInterceptor extends AbstractSecurity
Interceptor implements Filter {
    private static final Logger logger = Logger.getLogger(CustomFilterSecu
rityInterceptor.class);
    private FilterInvocationSecurityMetadataSource securityMetadataSource;

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
        FilterInvocation fi = new FilterInvocation(request, response, chain);
        logger.debug("====="+fi.getRequestUrl());
        invoke(fi);
    }

    public void invoke(FilterInvocation fi) throws IOException, ServletException {
        InterceptorStatusToken token = super.beforeInvocation(fi);
        try {
            fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
        } catch (Exception e) {
            logger.error(e.getMessage());
```

```
        } finally {
            super.afterInvocation(token, null);
        }
    }
    }
    ......
```

5.4.3　权限配置资源管理器

权限配置资源管理器实现了 Spring Security 的 FilterInvocationSecurityMetadataSource，它在启动时导入代码清单 5-4 的权限配置列表。如代码清单 5-21 所示，权限配置资源管理器为权限决断器实时提供支持，判断用户访问的资源是否在受保护的范围之内。

代码清单 5-21　权限配置资源管理器

```
    public class CustomSecurityMetadataSource implements FilterInvocationSecur
ityMetadataSource{
        private static final Logger logger = Logger.getLogger(CustomSecurityMe
tadataSource .class);

        private Map<String, Collection<ConfigAttribute>> resourceMap = null;
        private PathMatcher pathMatcher = new AntPathMatcher();

        private String urlroles;

        @Override
        public Collection<ConfigAttribute> getAllConfigAttributes() {
            return null;
        }

        public CustomSecurityMetadataSource  (String urlroles) {
            super();
            this.urlroles = urlroles;
            resourceMap = loadResourceMatchAuthority();
        }

        private Map<String, Collection<ConfigAttribute>> loadResourceMatchAuthority() {

            Map<String, Collection<ConfigAttribute>> map = new HashMap<String,
Collection<ConfigAttribute>>();

            if(urlroles != null && !urlroles.isEmpty()){
                String[] resouces = urlroles.split(";");
                for(String resource : resouces){
```

```
                String[] urls = resource.split("=");
                String[] roles = urls[1].split(",");
                Collection<ConfigAttribute> list = new ArrayList<ConfigAttri
bute>();
                for(String role : roles){
                    ConfigAttribute config = new SecurityConfig(role.trim());
                    list.add(config);
                }
                //key: url, value: roles
                map.put(urls[0].trim(), list);
            }
        }else{
            logger.error("'securityconfig.urlroles' must be set");
        }

        logger.info("Loaded UrlRoles Resources.");
        return map;
    }

    @Override
    public Collection<ConfigAttribute> getAttributes(Object object)
            throws IllegalArgumentException {
        String url = ((FilterInvocation) object).getRequestUrl();

        logger.debug("request url is " + url);

        if(resourceMap == null)
            resourceMap = loadResourceMatchAuthority();

        Iterator<String> ite = resourceMap.keySet().iterator();
        while (ite.hasNext()) {
            String resURL = ite.next();
            if (pathMatcher.match(resURL,url)) {
                return resourceMap.get(resURL);
            }
        }
        return resourceMap.get(url);
    }
    ......
}
```

5.4.4 权限管理决断器

权限管理的关键部分就是决断器，它实现了 Spring Security 的 AccessDecision-

Manager，重载了 decide 函数，使用了自定义的决断管理，如代码清单 5-22 所示。在用户访问受保护的资源时，决断器判断用户拥有的角色中是否对该资源具有访问权限，如果没有权限将被拒绝访问，并返回错误提示。

代码清单 5-22　权限管理决断器

```
public class CustomAccessDecisionManager implements AccessDecisionManager {
    private static final Logger logger = Logger.getLogger(CustomAccessDeci
sionManager.class);

    @Override
    public void decide(Authentication authentication, Object object, Collec
tion<ConfigAttribute> configAttributes)throws AccessDeniedException, Insufficie
ntAuthenticationException {
        if (configAttributes == null) {
            return;
        }

        //config urlroles
        Iterator<ConfigAttribute> iterator = configAttributes.iterator();

        while (iterator.hasNext()) {
            ConfigAttribute configAttribute = iterator.next();
            //need role
            String needRole = configAttribute.getAttribute();
            //user roles
            for (GrantedAuthority ga : authentication.getAuthorities()) {
                if (needRole.equals(ga.getAuthority())) {
                    return;
                }
            }
            logger.info("need role is " + needRole);
        }
        throw new AccessDeniedException("Cannot Access!");
    }
    ......
}
```

5.5　根据权限设置链接

对于权限管理，我们可能希望，在一个用户访问的界面中，不是等到用户单击了一个超链接之后，才来判断用户有没有这个权限（虽然这种设计是必须的），而是按照用户

拥有的权限来设置一个用户可以访问的超链接。这样的设计对于用户体验来说，显得更加友好。

以管理后台中用户管理的例子来说明如何实现根据权限来设置链接。如代码清单 5-23 所示，在打开用户管理主页的控制器中，读取了当前用户的权限配置，然后根据这个用户的权限列表来判断这个用户是否拥有新建、修改和删除等权限，最后把这些权限通过变量传给页面，由页面负责根据权限来设置用户可用的超链接。其中，newrole、editrole 和 deleterole 分别表示新建、修改和删除权限的判断值。

代码清单 5-23 在控制器中获取用户权限

```
@Value("${securityconfig.urlroles}")
    private String urlroles;

    @RequestMapping("/index")
    public String index(ModelMap model, Principal user) throws Exception{
        Authentication authentication = (Authentication)user;
        List<String> userroles = new ArrayList<>();
        for(GrantedAuthority ga : authentication.getAuthorities()){
            userroles.add(ga.getAuthority());
        }

        boolean newrole=false,editrole=false,deleterole=false;
        if(!StringUtils.isEmpty(urlroles)) {
            String[] resouces = urlroles.split(";");
            for (String resource : resouces) {
                String[] urls = resource.split("=");
                if(urls[0].indexOf("new") > 0){
                    String[] newroles = urls[1].split(",");
                    for(String str : newroles){
                        str = str.trim();
                        if(userroles.contains(str)){
                            newrole = true;
                            break;
                        }
                    }
                }else if(urls[0].indexOf("edit") > 0){
                    String[] editoles = urls[1].split(",");
                    for(String str : editoles){
                        str = str.trim();
                        if(userroles.contains(str)){
                            editrole = true;
                            break;
                        }
                    }
```

```
            }else if(urls[0].indexOf("delete") > 0){
                String[] deleteroles = urls[1].split(",");
                for(String str : deleteroles){
                    str = str.trim();
                    if(userroles.contains(str)){
                        deleterole = true;
                        break;
                    }
                }
            }
        }
    }

    model.addAttribute("newrole", newrole);
    model.addAttribute("editrole", editrole);
    model.addAttribute("deleterole", deleterole);

    model.addAttribute("user", user);
    return "user/index";
}
```

在用户管理的主页视图中有一个“新增”超链接，可以通过控制器传递过来的 newrole 值来判断这个用户对这个链接有没有权限，从而决定这个链接能不能显示出来，提供给用户使用，代码如下：

```
<div class="newBtnBox" th:if="${newrole}">
<a id="addUserInf" class="blueBtn-62X30" href="javascript:void(0)">新增 </a>
</div>
```

而对于修改和删除的权限，因为页面的数据是从 js 中生成的，所以可以在生成用户列表的程序段中判断 editrole 和 deleterole，从而决定是否提供这两个功能的链接，如代码清单 5-24 所示。

代码清单 5-24　在 js 中根据用户权限设置链接

```
//填充分页数据
function fillData(data){
    var editrole = $("#editrole").val();
    var deleterole = $("#deleterole").val();

    var $list = $('#tbodyContent').empty();
    $.each(data,function(k,v) {
        var html = "";
        html += '<tr> ' +
            '<td>' + (v.id == null ? '' : v.id) + '</td>' +
```

```
                '<td>' + (v.name == null ? '' : v.name) + '</td>' +
                '<td>' + (v.email == null ? '' : v.email) + '</td>' +
                '<td>' + (v.createdate == null ? '' : getSmpFormatDateByLong(v.
createdate, true)) + '</td>';
            html += '<td><a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"showDetail(\'' + v.id + '\')">查看</a>';

            if (editrole == 'true')
                html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"edit(\'' + v.id + '\')">修改</a>';

            if(deleterole == 'true')
                html += '<a class="c-50a73f mlr-6" href="javascript:void(0)" onclick=
"del(\''+ v.id+'\')">删除</a>';

            html +='</td></tr>' ;

            $list.append($(html));
        });
    }
```

其中的“修改”和“删除”权限的判断值，即代码中的 editrole 和 deleterole，是在导入用户管理的主页时使用隐藏的输入框这种方式传递进来的，代码如下：

```
    <input type="hidden" name="editrole" id="editrole" th:value="${editrole}"/>
    <input type="hidden" name="deleterole" id="deleterole" th:value=
"${deleterole}"/>
```

上面这种根据权限设置链接的设计，只是在一个局部操作界面上实现，在实际应用中，可以通过统筹规划全局视图，在全局的角度中实现这种设计。

5.6 运行与发布

本章实例工程的完整代码可以通过 IDEA 直接从 GitHub 中检出：https://github.com/chenfromsz/spring-boot-security.git。实例工程中包含两个模块：mysql 和 web，其中 mysql 模块提供数据库管理功能，web 模块集成了安全管理和一个数据管理后台的功能，即用户登录后可以对用户、部门和角色等各个对象的数据进行管理。

5.6.1 系统初始化

为了初始化一个能够登录系统的用户，我们在工程模块 mysql 中编写了一个 JUint

测试程序，用来生成一个具有所属部门和拥有角色的用户，如代码清单 5-25 所示。测试程序执行时，将初始化数据库，并生成一个部门和一个角色，同时创建一个初始用户，用户名和密码都为 user，这个用户默认具有管理员的权限。

代码清单 5-25　系统初始化测试程序

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {JpaConfiguration.class})
public class MysqlTest {
    @Autowired
    UserRepository userRepository;
    @Autowired
    DepartmentRepository departmentRepository;
    @Autowired
    RoleRepository roleRepository;

    @Before
    public void initData(){
        userRepository.deleteAll();
        roleRepository.deleteAll();
        departmentRepository.deleteAll();

        Department department = new Department();
        department.setName("开发部");
        departmentRepository.save(department);
        Assert.notNull(department.getId());

        Role role = new Role();
        role.setName("admin");
        roleRepository.save(role);
        Assert.notNull(role.getId());

        User user = new User();
        user.setName("user");
        BCryptPasswordEncoder bpe = new BCryptPasswordEncoder();
        user.setPassword(bpe.encode("user"));
        user.setCreatedate(new Date());
        user.setDepartment(department);
        userRepository.save(user);
        Assert.notNull(user.getId());
    }

    @Test
    public void insertUserRoles(){
        User user = userRepository.findByName("user");
```

```
        Assert.notNull(user);

        List<Role> roles = roleRepository.findAll();
        Assert.notNull(roles);
        user.setRoles(roles);
        userRepository.save(user);
    }
}
```

这样，就可以在 IDEA 的 Edit Configuration 中增加一个 JUint 测试配置项目，模块选择 mysql，工作目录选择 mysql 模块所在的工程根目录，测试类选择上面的测试程序，并将配置保存为 MysqlTest。

然后在 MySQL 服务器中创建一个 test 数据库，并在测试程序所在目录中打开配置类 JpaConfiguration 的实现代码，配置数据源中的 url、username、password，如代码清单 5-26 所示。

代码清单 5-26 测试程序的 JpaConfiguration 数据源配置

```
@Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/test?characterEncoding=
utf8");
        dataSource.setUsername("root");
        dataSource.setPassword("12345678");

        return dataSource;
    }
```

最后运行测试项目 MysqlTest，运行成功后生成一个初始用户，用户名和密码为 user，并且该用户的所属部门为“开发部”，拥有一个管理员角色为 admin。

5.6.2 系统运行与发布

首先，在 web 模块的配置文件 application.yml 中配置连接 MySQL 服务器的数据源，其他 JPA 和安全配置可以保持不变。

如果在 IDEA 中运行应用，可以在 IDEA 的 Edit Configuration 中增加一个 Spring Boot 配置项目，模块选择 web，工作目录选择 web 模块所在的工程根目录，主程序选择 com.test.web.WebApplication，并将配置项目保存为 web。

然后运行配置项目 web 即可启动 web 应用，启动成功后，在浏览器中输入网址 http://localhost 访问应用。

在出现的登录界面中，输入上面生成的用户名和密码 user，并输入正确的验证码，即可登录系统。登录系统后，可对用户、部门和角色进行管理。

如果要发布应用，既可以在 IDEA 中增加一个 Maven 配置，也可以打开一个命令行窗口，将目录切换到工程的根目录，然后执行下列指令来完成。

```
mvn clean package
```

注意　使用上面的发布指令将会自动调用 MysqlTest 测试程序，这将删除数据库的所有资料，执行初始化操作，然后创建一个初始用户，用户名和密码都为 user，并且具有管理员的权限。如果不想自动调用测试程序，可在上面指令中加上参数：-D skipTests。

打包成功后可在命令行窗口中，运行下列指令启动系统（假如当前目录为工程根目录）。

```
java -jar web/target/web-1.0-SNAPSHOT.jar
```

5.7　小结

本章使用 Spring Security，实现了 Web 应用系统的安全管理功能，即用户认证和权限管理等功能。使用 Spring Security 省略了很多安全管理的设计和实现的工作，同时引用 Spring Security 的功能，使用一些自定义的设计，又让安全管理设计增加了很多灵活性，例如，可以设计出更加漂亮的登录界面、更加简便的权限管理措施和策略等。

Spring Security 安全管理功能完善而且强大，那么对于分布式应用环境，又将怎样使用呢？下一章使用 Spring Security 结合 OAuth2 协议来设计分布式应用环境中的单点登录。

第二部分 *Part 2*

分布式应用开发

- 第 6 章　Spring Boot SSO
- 第 7 章　使用分布式文件系统
- 第 8 章　云应用开发
- 第 9 章　构建高性能的服务平台

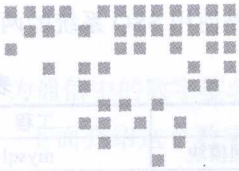
这一部分介绍分布式应用系统的开发及其怎么构建一个高性能的服务平台。

第 6 章介绍在分布应用系统中怎样进行安全管理，并使用 Spring Security 结合 OAuth2 设计一个 SSO 管理系统。

第 7 章介绍如何在 Spring Boot 中使用分布式文件管理系统，同时使用定制方式和富文本编辑器方式演示了文件上传的功能，还介绍了怎样建立和管理本地文件库。

第 8 章使用 Spring Cloud 云应用开发工具集，介绍了配置管理、发现服务和监控服务的使用，以及如何使用动态路由和断路器的功能，创建高可用的微服务应用。

第 9 章介绍使用 Docker 引擎和 docker-compose 工具来发布应用和管理服务，以及如何构建一个高性能的服务平台和怎样使用 Docker 实施负载均衡。



第 6 章 Chapter 6

Spring Boot SSO

一个企业级的应用系统可能存在很多应用系统，每个应用系统都需要设计安全管理，即实现用户的认证和访问授权，但是不可能为每一个应用系统都设计一套安全管理，这样不但耗时耗力，而且要做重复的工作，也不适宜建立统一的用户中心。这就需要使用单点登录（Single Sign On，SSO）的方式来建立一个登录认证系统，并且实现对用户的统一管理。对于一个开放平台来说，SSO 也能为合作伙伴提供用户的身份认证和授权管理。

将使用第 5 章的安全设计，再略加以扩展来建立一个 SSO 管理系统。这里介绍的 SSO 管理系统，是在使用 Spring Security 安全管理的基础上，再结合 OAuth2 认证授权协议来实现的，它不但适用于大型的分布式管理系统，也适用于为第三方提供统一的用户管理和认证的平台。

6.1 模块化设计

本章的实例工程由于涉及的功能较多，将按照表 6-1，对实例工程实行模块化管理。其中，每个模块都是一个独立的项目，数据库管理模块为其他模块提供数据管理支持，安全配置模块为客户端提供安全配置和授权管理支持，登录认证模块提供单点登录认证（即 SSO）功能，共享资源模块为客户端提供登录用户需要的一些共享资源，两个

客户端应用是使用 SSO 系统的两个实例。

表 6-1 实例工程模块列表

项目	工程	类型	功能
数据库管理模块	mysql	程序集成	数据库管理
安全配置模块	security	程序集成	安全策略配置和权限管理
登录认证模块	login	Web 应用	SSO 登录认证（使用端口：80）
共享资源模块	resource	Web 应用	共享资源（使用端口：8083）
客户端应用 1	web1	Web 应用	客户端 1（使用端口：8081）
客户端应用 2	web2	Web 应用	客户端 2（使用端口：8082）

使用模块化设计可以提高代码的复用性，避免重复开发，从而提高开发速度和工作效率。例如，实例工程的数据库管理模块和安全配置模块能够被其他模块共用，从而减少了大部分重复的工作。

其中，数据库管理模块 mysql 与第 5 章的 mysql 模块的功能完全相同，它为其他各个模块提供了数据管理功能，同样具有部门、用户和角色三个实体，并且提供了对这三个实体对象的增删查改等操作的功能。

6.2 登录认证模块

如果只是本地的登录认证，只要使用 Spring Security 就足够了。由于使用 SSO 实现了远程的登录认证功能，所以在登录认证系统中，需要增加 OAuth2 协议，让它可以支持第三方应用的认证和授权。

登录认证系统将建立一个用户中心，对使用 SSO 服务的每一个应用系统，提供统一的用户管理。而对于一个用户来说，使用任何一个应用系统，都可以通过 SSO 的 OAuth2 协议进行认证和授权确认。

6.2.1 使用 OAuth2

要使用 OAuth2，在登录认证模块和安全配置模块中都要在工程的 Maven 依赖管理中增加 OAuth2 的依赖配置，如代码清单 6-1 所示。

代码清单 6-1 OAuth2 依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

6.2.2 创建数字证书

在 OAuth2 的认证服务端中，需要一个数字证书，为通信中的数字签名等功能提供支持。这个数字证书可以使用 Java 的 keystore 来生成。下面介绍这个数字证书的生成过程，在实例工程中已经具有这个证书，不用重新生成。

在 Windows 操作系统中打开一个命令行窗口，使用下列的指令可以生成一个数字证书：

```
C:\Users\Alan>keytool -genkey -keystore keystore.jks -alias tycoonclient
-keyalg RSA
```

执行这个指令的操作过程如下：

```
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
    [Unknown]:  localhost
您的组织单位名称是什么？
    [Unknown]:  test
您的组织名称是什么？
    [Unknown]:  test
您所在的城市或区域名称是什么？
    [Unknown]:  sz
您所在的省 / 市 / 自治区名称是什么？
    [Unknown]:  gd
该单位的双字母国家 / 地区代码是什么？
    [Unknown]:  cn
CN=localhost, OU=test, O=test, L=sz, ST=gd, C=cn 是否正确？
    [否]:  y

输入 <tycoonclient> 的密钥口令
        （如果和密钥库口令相同，按回车）：
```

在上面操作的过程中，输入的密码是 tc123456，证书的别名设定为 tycoonclient，证书的文件保存为 keystore.jks。

然后，将生成的证书文件拷贝到登录认证模块的 resources 文件夹中，并在 OAuth2 配置中设定其相应的参数。

6.2.3 认证服务端配置

登录认证模块实现了 SSO 认证和授权服务的功能，在这里必须对 OAuth2 的认证和授权服务，以及对 Spring Security 的安全管理策略等进行一些相关的设计和配置，为

使用 SSO 的客户端提供认证和授权的管理功能。

1. OAuth2 服务端配置

在登录认证模块中，编写一个 OAuthConfigurer 配置类程序，如代码清单 6-2 所示，它继承了 AuthorizationServerConfigurerAdapter。其中，使用注解 @EnableAuthorizationServer 来启用 OAuth2 的认证服务器功能。在 JwtAccessTokenConverter 方法中使用上面生成的数字证书：keystore.jks，并设置了密码和别名等参数。在重载的 configure 方法中设定 OAuth2 的客户端 ID 为 ssoclient，密钥为 ssosecret，这将在使用 SSO 的客户端的配置中用到。另外，注意“autoApprove (true)”这行代码设定了自动确认授权，这样登录用户登录后，不再需要进行一次授权确认操作。

代码清单 6-2 OAuth2 服务端配置

```
@Configuration
@EnableAuthorizationServer
public class OAuthConfigurer extends AuthorizationServerConfigurerAdapter {

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
        KeyPair keyPair = new KeyStoreKeyFactory(new ClassPathResource(
                "keystore.jks"), "tc123456".toCharArray()).getKeyPair("tycoon
client");
        converter.setKeyPair(keyPair);
        return converter;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients)
            throws Exception {
        clients.inMemory().withClient("ssoclient").secret("ssosecret")
                .autoApprove(true)
                .authorizedGrantTypes("authorization_code", "refresh_token")
.scopes("openid");
    }
    ...
}
```

2. Spring Security 服务端配置

因为认证服务器的 Spring Security 安全策略配置与客户端的安全策略配置不同，所

以它没有使用工程中安全配置模块中的配置，而是单独使用一个配置类来实现，如代码清单 6-3 所示。这里有点像第 5 章的安全策略配置，依然提供了记住用户登录状态的功能，这样当用户选择记住登录状态登录后，只要用户不执行退出，在记住登录状态的有效期内，重新打开授权的链接时就可以不用再次登录。登录页面设定还是使用“/login”。但是这里没有针对角色的一些权限管理配置，这是因为在登录认证模块中只提供了登录认证功能，并不提供其他访问链接，所以这里不需要配置一些链接的角色权限管理。

代码清单 6-3　认证服务器的安全策略配置

```
    @Configuration
    @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
    public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
        @Autowired
        private CustomUserDetailsService customUserDetailsService;
        @Autowired @Qualifier("dataSource")
        private DataSource dataSource;

        @Override
        protected void configure(AuthenticationManagerBuilder auth)
                throws Exception {
            auth.userDetailsService(customUserDetailsService).passwordEncoder(
passwordEncoder());
            //remember me
            auth.eraseCredentials(false);
        }

        @Override
        protected void configure(HttpSecurity http) throws Exception {
    http.formLogin().loginPage("/login").permitAll().successHandler(loginSuccess
Handler())
                    .and().authorizeRequests()
                    .antMatchers("/images/**", "/checkcode", "/scripts/**", "/styles/
**").permitAll()
                    .anyRequest().authenticated()
                    .and().sessionManagement().sessionCreationPolicy(SessionCrea
tionPolicy.NEVER)
                    .and().exceptionHandling().accessDeniedPage("/deny")
                    .and().rememberMe().tokenValiditySeconds(86400).tokenRepository
(tokenRepository());
        }
        ......
    }
```

6.3 安全配置模块

安全配置模块集成了SSO客户端的安全策略配置和权限管理功能，可以供使用SSO的客户端使用。代码清单6-4是客户端的安全策略配置，其中，注解@EnableOAuth2Sso将应用标注为一个SSO客户端，重载的configure方法使用了HttpSecurity来配置一些安全管理策略。注意这里没有登录链接的配置，因为登录认证的功能已经交给OAuth2处理了。另外，CustomFilterSecurityInterceptor设定了使用自定义的权限管理过滤器，这个功能还是与第5章的设计一样，这里不再赘述。

代码清单6-4 客户端安全策略配置

```
@Configuration
@EnableOAuth2Sso
@EnableConfigurationProperties(SecuritySettings.class)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private SecuritySettings settings;

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
                .antMatcher("/**").authorizeRequests()
                .antMatchers(settings.getPermitall().split(",")).permitAll()
                .anyRequest().authenticated()
                .and().csrf().requireCsrfProtectionMatcher(csrfSecurityRequest
Matcher())
                .csrfTokenRepository(csrfTokenRepository()).and()
                .addFilterAfter(csrfHeaderFilter(), CsrfFilter.class)
                .logout().logoutUrl("/logout").permitAll()
                .logoutSuccessUrl(settings.getLogoutsuccssurl())
                .and()
                .exceptionHandling().accessDeniedPage(settings.getDeniedpage());
    }

    @Bean
    public CustomFilterSecurityInterceptor customFilter() throws Exception{
        CustomFilterSecurityInterceptor customFilter = new CustomFilterSecurity
Interceptor();
        customFilter.setSecurityMetadataSource(securityMetadataSource());
        customFilter.setAccessDecisionManager(accessDecisionManager());
        customFilter.setAuthenticationManager(authenticationManager);
```

```
        return customFilter;
    }
    ......
}
```

6.4　SSO 客户端

SSO 客户端要使用安全配置模块的功能，需要在工程的 Maven 依赖管理中，增加一个依赖配置，如代码清单 6-5 所示。

代码清单 6-5　引用安全配置模块的依赖配置

```
<dependency>
            <groupId>springboot.demo</groupId>
            <artifactId>security</artifactId>
            <version>${project.version}</version>
</dependency>
```

6.4.1　客户端配置

当客户端引用安全配置模块之后，就必须在配置文件 application.yml 中进行一些相关的配置，才能正常使用。代码清单 6-6 是一个 SSO 客户端的配置，它包含两方面的内容，其中 security 是 OAuth2 的配置，securityconfig 是 Spring Security 的配置。

在 OAuth2 的配置中，loginPath 是一个登录的链接地址，clientId 和 clientSecret 是由 SSO 认证服务器提供的客户端 ID 和密钥，accessTokenUri 是取得令牌的链接地址，userAuthorizationUri 是用户授权确认的链接地址，keyUri 是当客户端被指定为资源服务器时所用的令牌链接地址。

Spring Security 的配置中是设计的一些自定义配置参数，它将被安全管理策略配置类调用，其中 logoutsuccssurl 是一个登出成功的链接地址，其他配置参数与第 5 章的安全管理配置基本相同。

代码清单 6-6　SSO 客户端配置

```
security:
  ignored: /favicon.ico,/scripts/**,/styles/**,/images/**
  sessions: ALWAYS
  oauth2:
      sso:
        loginPath: /login
```

```
    client:
      clientId: ssoclient
      clientSecret: ssosecret
      accessTokenUri: http://localhost/oauth/token
      userAuthorizationUri: http://localhost/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      jwt:
        keyUri: http://localhost/oauth/token_key

securityconfig:
  logoutsuccssurl: /tosignout
  permitall: /rest/**,/bb**
  deniedpage: /deny
  urlroles: /**/new/** = admin;
            /**/edit/** = admin,editor;
            /**/delete/** = admin
```

6.4.2 登录登出设计

登录登出的设计，虽然在 Spring Security 中已经实现，但是对于使用 SSO 的客户端来说，必须进行一些合理的调整，否则如果设计不当，就有可能出现无法正常退出或者登录失败的情况。

代码清单 6-7 是客户端的一个登录控制器设计，它使用一个重定向链接“redirect:/#/”来刷新当前访问页面，从而触发系统检查用户的授权状态，如果用户未被授权，则引导用户到登录认证服务器中登录。

代码清单 6-7 客户端登录控制器

```
@RequestMapping("/login")
public String login() {
    return "redirect:/#/";
}
```

代码清单 6-8 是客户端的一个登出设计，退出时首先通过用户确认，然后使用 POST 方式执行表单 logoutform 的退出提交请求，这个请求已由 Spring Security 实现，执行一些清除会话和登录状态等操作，并将当前操作界面重定向到登出成功页面上。

这里需要注意的是，用户这时只是退出当前客户端而已，并没有在 SSO 服务端中执行过退出请求，也就是说，在 SSO 认证服务端中，还保存着用户的登录状态。如果这时返回原来的客户端，或者访问其他有授权的客户端，都不会要求用户登录并能正常

访问。为了能达真正的退出，还必须在 SSO 服务端中再执行一次退出请求。这个退出请求必须由程序来处理，而不能要求用户转到 SSO 服务端中再执行一次退出。

代码清单 6-8　客户端登出设计

```
<a href="javascript:void(0)" id="logout">[ 退出 ]</a>
......
<form th:action="@{/logout}" method="post" id="logoutform">
</form>
<script type="text/javascript">
    $(function () {
        $("#logout").click(function () {
            if (confirm('您确定退出吗？ ')) {
                $("#logoutform").submit();
            }
        });
    });
</script>
```

在客户端配置中有一个成功退出的链接地址，当用户在客户端中成功退出时，将被重定向到这个链接地址。代码清单 6-9 是这个链接的页面设计，它只做一件简单的事情，即在当前页面中做一个跳转链接，转到 SSO 服务端中执行退出请求。

代码清单 6-9　跳转到 SSO 服务端中执行登出

```
<script>
    function to_sso(){
        location.href = "http://localhost/signout";
    }
</script>
<body onload="to_sso()">
</body>
```

代码清单 6-10 是 SSO 服务端的登出控制器设计，只有请求这个链接，才能让用户完全退出当前的登录状态。程序中使用 request.logout() 来请求 Spring Security 执行退出请求，然后返回到 SSO 的登录界面，以刷新当前的页面状态。

代码清单 6-10　SSO 服务端登出控制器

```
@RequestMapping("/signout")
    public String signout(HttpServletRequest request) throws Exception{
        request.logout();
        return "tologin";
    }
```

代码清单 6-11 是接收上面的请求后，返回的一个页面设计，它同样也只做一件简单的事情，即将当前页面跳转到用户登录界面上。

代码清单 6-11 跳转到登录界面

```
<script>
    function new_window(){
        location.href = "/login";
    }
</script>
<body onload="new_window()">
</body>
```

这样，用户不管在哪个客户端中执行退出，通过上面的跳转，最终都将被引导到 SSO 服务端的登录界面上。通过这些流程的处理，用户的一个退出请求才能彻底退出登录状态。当然，上面这些跳转对于用户来说，是完全透明的。

用户打开任何一个客户端的页面进行登录，登录成功后将被引导到最初打开的页面上。如果用户直接在 SSO 认证服务端上登录，必须有一个成功登录后的默认主页，这个页面可以配置一些其他客户端的导航链接。因为把登录成功的默认主页设计放置在客户端 1 的主页上，所以如果用户从 SSO 服务器中直接登录，就可以在 SSO 服务器的主页上做一个跳转，如代码清单 6-12 所示，将跳转到客户端 1 的主页上。

代码清单 6-12 登录成功的默认链接设计

```
<script>
    <!--
    function new_window(){
        location.href = "http://localhost:8081/";
    }
    //-->
</script>

<!-- onload="new_window()"-->
<body onload="new_window()">
...... 请稍候
</body>
```

6.5 共享资源服务

实例工程的共享资源模块，可以为已经授权的用户提供一些共享信息服务。代码清

单 6-13 是共享资源模块的主程序，它使用注解 @EnableResourceServer 来标注这个应用是一个资源服务器。

代码清单 6-13　资源服务器主程序

```
@SpringBootApplication
@EnableResourceServer
@ComponentScan(basePackages = "com.test")
public class ResourceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ResourceApplication.class, args);
    }
}
```

一个应用被标注为资源服务器后，在浏览器中就不能直接访问，如果在浏览器上打开这样的客户端，将只能看到如图 6-1 所示的提示信息。

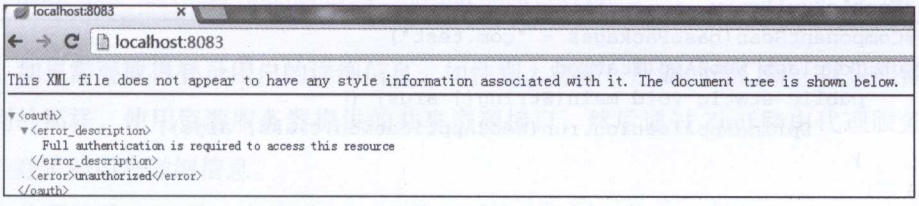


图 6-1　在浏览器中打开资源服务器的情况

6.5.1　提供共享资源接口

启用资源服务器功能之后，就能够对外提供资源信息服务。代码清单 6-14 是一个共享登录用户信息的接口设计，这是提供了一个“/user”链接的控制器，程序中通过 Principal 取得登录用户的用户名，然后通过用户名在数据库中查出用户的详细信息，最后返回包含用户信息的一个 Map 对象。

代码清单 6-14　共享用户信息接口设计

```
@Autowired
    private UserRepository userRepository;

    @RequestMapping("/user")
    public Map<String, Object> user(Principal puser) {
        User user = userRepository.findByName(puser.getName());
        Map<String, Object> userinfo = new HashMap<>();
        userinfo.put("id", user.getId());
```

```
        userinfo.put("name",user.getName());
        userinfo.put("email", user.getEmail());
        userinfo.put("department",user.getDepartment().getName());
        userinfo.put("createdate", user.getCreatedate());
        return userinfo;
    }
```

6.5.2 使用共享资源

在客户端中要使用资源服务器的共享信息，可以使用 spring-cloud-zuul 提供的一个路由服务来实现。代码清单 6-15 是客户端应用 2 的主程序，它使用注解 @EnableZuulProxy 来启用 Zuul 路由代理服务。

代码清单 6-15 客户端应用 2 的主程序

```
@SpringBootApplication
@EnableZuulProxy
@ComponentScan(basePackages = "com.test")
public class Web2Application {
    public static void main(String[] args) {
        SpringApplication.run(Web2Application.class, args);
    }
}
```

在工程配置文件 application.yml 中使用如代码清单 6-16 所示的配置，配置一个路由资源，其中 path 设定资源的访问路径，url 指定路由的服务方。

代码清单 6-16 使用资源服务的路由配置

```
zuul:
    routes:
        resource:
            path: /resource/**
            url: http://localhost:8083
            stripPrefix: true
            retryable: true
```

这样就可以在客户端应用 2 中使用如下的链接进行访问：

```
http://localhost:8082/resource/user
```

或者通过程序使用如下的 Ajax 方式获取数据：

```
$.get('./resource/user',{ts:new Date().getTime()},function(data)
```

6.5.3　查询登录用户的详细信息

在单独使用 Spring Security 安全管理的应用中，只要在控制器中使用 Principal，就能取得用户的完整信息，或者使用如代码清单 6-17 所示的代码，也能很容易地获取登录用户的详细信息。但是，使用 SSO 之后，这种方法就不能适用了，这时如果使用 getDetails() 将返回一个空值，而在控制器中使用 Principal 也只能返回登录用户的用户名、用户拥有的角色和登录令牌等信息而已，用户的其他信息如性别、邮箱等将不能取得。这是 OAuth2 基于安全的考虑而设计的，因为 SSO 涉及了第三方的应用请求，所以它保护了登录用户的隐私信息。

代码清单 6-17　查看登录用户的详细信息

```
    Authentication authentication = SecurityContextHolder.getContext().getAuthen
tication();
    User user = (User)authentication.getDetails();
```

如果需要取得登录用户的详细信息，如性别、邮箱、所属的部门等，就只能像前面提到的那样，使用资源服务器提供的共享资源接口，然后通过 Zuul 路由代理服务获取已经登录的用户详细信息。

代码清单 6-18 是一个使用 Ajax 获取登录用户的详细信息的例子。

代码清单 6-18　从资源服务器中获取登录用户的详细信息

```
    function getuserinfo(){
        $.get('./resource/user',{ts:new Date().getTime()},function(data){
            var $list = $('#tbodyContent').empty();
                var html = "" ;
                html += '<tr> ' +
                    '<td>'+ (data.id==null?'':data.id) +'</td>' +
                      '<td>'+ (data.name==null?'':data.name) +'</td>' +
                      '<td>'+ (data.email==null?'':data.email) +'</td>' +
                      '<td>'+ (data.department==null?'' :data.department) +'</td>' +
                      '<td>'+ (data.createdate==null?'': getSmpFormatDateByLong(data.
createdate,true)) +'</td>';
                html +='</tr>' ;
                $list.append($(html));
        });
    }
```

最后完成的用户信息查询的效果图如图 6-2 所示。

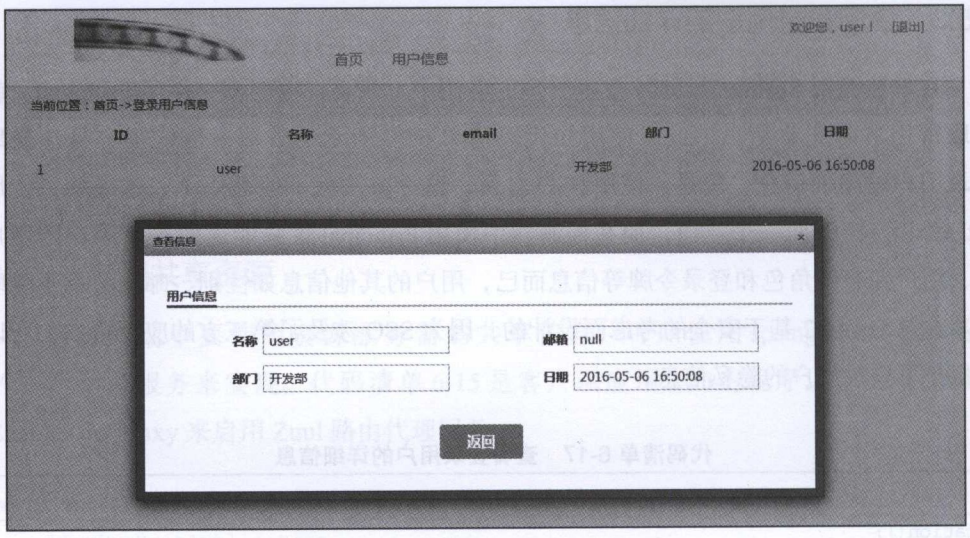


图 6-2 获取用户信息示例

6.6 运行与发布

本章实例工程的完整代码可以通过 IDEA 在 GitHub 中检出：https://github.com/chen-fromsz/spring-boot-sso.git。

检出工程后，在本地的 MySQL 服务器中创建一个数据库 test，并运行下列查询指令设定使用数据库的用户名和密码。

```
grant all privileges on test.* to 'root'@'localhost' identified by '12345678';
```

然后打开 IDEA 的 Edit Configuration 对话框，按下列步骤增加配置：

1）在数据库管理模块中增加一个 JUint 测试配置，运行 MysqlTest 测试程序，用来生成默认的登录用户，最终生成的用户名和密码均为 user。

2）在登录认证模块增加一个 Spring Boot 配置，用来运行 LoginApplication。

3）在共享资源模块增加一个 Spring Boot 配置，用来运行 ResourceApplication。

4）在客户端应用 1 模块增加一个 Spring Boot 配置，用来运行 Web1Application。

5）在客户端应用 2 模块增加一个 Spring Boot 配置，用来运行 Web2Application。

运行测试程序生成默认的登录用户后，可以按下列顺序运行各个应用：

1）运行登录认证服务。

2）运行共享资源服务。

3）运行客户端应用 1。

4）运行客户端应用 2。

各个应用启动完成后在浏览器中输入：http://localhost。

在登录界面上使用上述创建的默认用户登录，登录成功即进入客户应用 1 的系统首页，如图 6-3 所示。需要注意的是，在单机上运行上面 4 个应用需要耗费一定的内存。如果在各个应用的配置文件中，合理配置 IP 地址，也可以将 4 个应用发布在不同的机器上运行，其结果相同。

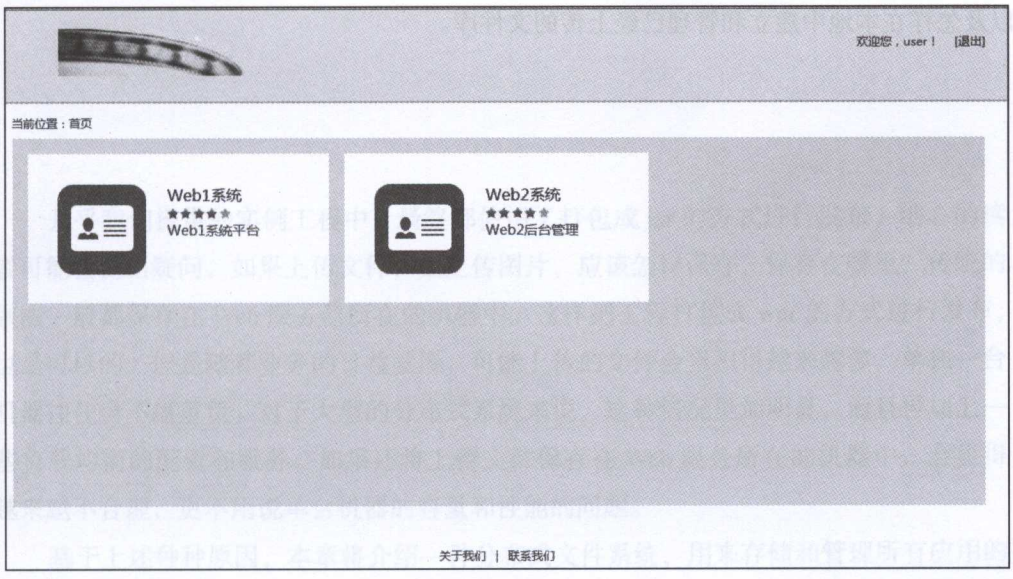


图 6-3　SSO 应用示例

如果需要进行发布打包，可以打开命令行窗口，将当前路径切换到工程根目录中，然后执行下列 Maven 指令：

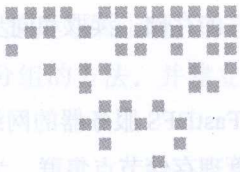
```
mvn clean package
```

注意　运行打包指令会默认调用测试程序，这将对数据库进行初始化，并生成一个具有管理员权限的默认用户。在上面指令中增加参数“-D skipTests”可以跳过测试。

6.7 小结

本章通过使用 Spring Security 的安全管理功能，结合使用 OAuth2 的认证授权协议，设计了一个具有统一用户管理中心的 SSO 管理系统，实现了能够为第三方应用提供登录认证和授权管理的功能，为企业级的分布式应用系统开发，提供了切实可行的应用实例。同时使用 spring-cloud-zuul 的路由功能，演示了如何通过 SSO 系统使用安全的共享资源。

系统的安全设计非常重要，但系统的访问性能更为重要，下一章将介绍使用分布式文件系统来提升应用系统的访问性能，并演示如何在分布式环境中使用图片上传功能，以及怎样在本地中建立和管理已经上传的文件库。



第 7 章 Chapter 7

使用分布式文件系统

几乎我们提供的实例工程中，最终都使用了打包成 jar 的方式进行发布，细心的读者可能会提出疑问，如果上传文件，如上传图片，应该怎样保存，保存在哪里？传统的做法一般都保存在 Web 服务器所在的机器中。这样把工程打包成 war 的方式进行发布，也是可以的。但是随着业务的日益发展，可能上传的文件会累积得越来越多，单独一台机器往往会不堪重负。对于大型的分布式系统来说，这种情况更加明显，而且再加上一些负载均衡的配置和服务，如果还将上传文件保存在 Web 服务所在的机器中，会显得越来越不合理，更不用说单台机器的容量和性能的问题。

基于上述种种原因，本章将介绍一种分布式文件系统，用来存储和管理所有应用的上传文件。

在诸多分布式的文件系统中，FastDFS 是比较优秀的分布式文件系统。FastDFS 是一个完全开源的分布式文件系统，使用比较简单方便，而且性能也很优秀，存储容量和访问性能可按需求进行线性横向扩展，即可以通过增加设备的方式实现动态扩容。

7.1 FastDFS 安装

FastDFS 分为服务端和客户端 API 两部分，服务端又分为跟踪器（Tracker）和存储节点（Storage）两个角色。跟踪器主要负责服务调度，起着负载均衡的作用。存储节点

主要负责文件的存储、读取和同步等功能。客户端 API 提供了文件的上传、下载和删除等操作方法。

典型的 FastDFS 服务器的网络拓扑结构如图 7-1 所示，客户端连接跟踪器服务器集群，跟踪器管理存储节点集群，为客户端提供可用的存储节点。

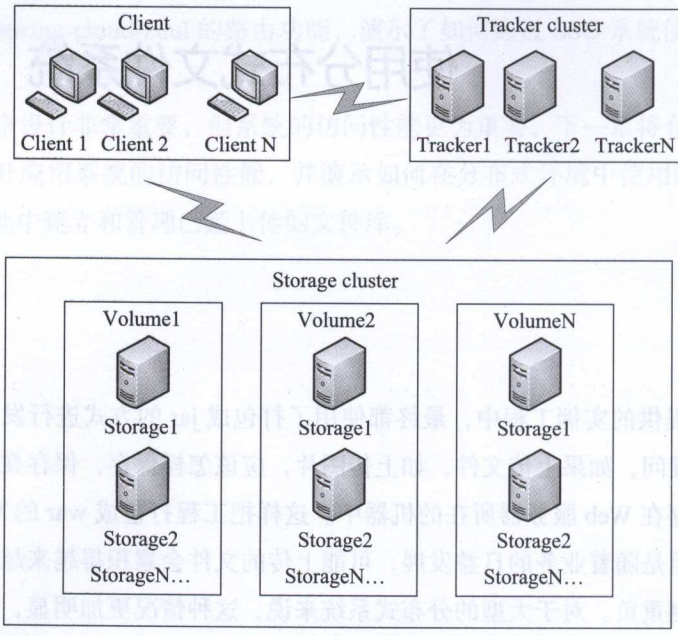


图 7-1 FastDFS 服务器网络拓扑

FastDFS 支持动态扩展，当资源快要耗尽时，可以通过增加新卷（或新组）的方法，增加更多的存储节点。

存储节点的文件使用了分卷（或分组）的组织方式，所以一个文件的标识由卷名（或组名）、路径和文件名等部分组成。

为了演示使用 FastFDS，使用虚拟机安装了三个 Linux 系统。表 7-1 列出了三个服务器的 IP 地址、安装的操作系统和各自的用途。

表 7-1 FastFDS 服务器列表

IP	系统	用途
192.168.1.214	CentOS 6.5	Tracker Server
192.168.1.215	CentOS 6.5	Storage Server
192.168.1.216	CentOS 6.5	Storage Server

下面将按照步骤说明安装的过程，由于使用 Nginx 提供文件的浏览访问功能，同时也需要安装 Nginx 服务，对于文件的组织方式使用了分组的方法，并建立了一个分组：group1。

7.1.1　下载安装包

登录 Tracker Server，使用下列指令切换到 opt 目录。

```
#cd  /opt
```

然后使用 wget 指令下载下列各个安装包，最后将下载的文件拷贝到其他两台服务器的相同目录中（注：检出实例工程后，在 doc 目录中包含下列安装包）。

1. 下载 FastDFS 5.01

```
#wget  http://jaist.dl.sourceforge.net/project/fastdfs/FastDFS%20Server%20Source%20Code/FastDFS%20Server%20with%20PHP%20Extension%20Source%20Code%20V5.01/FastDFS_v5.01.tar.gz
```

2. 下载 nginx 1.7.0

```
#wget http://nginx.org/download/nginx-1.7.0.tar.gz
```

3. 下载 fastdfs-nginx-module_v1.16

```
#wget http://jaist.dl.sourceforge.net/project/fastdfs/FastDFS%20Nginx%20Module%20Source%20Code/fastdfs-nginx-module_v1.16.tar.gz
```

7.1.2　安装服务

三台服务器上都要安装 FastDFS 和 Nginx，其中 FastFDS 使用了默认的安装配置，Nginx 使用了自定义的安装配置。

要执行安装指令，系统中必须具有编译环境，如果系统还没有编译环境，可以使用下列指令增加编译环境。

```
#yum -y install gcc gcc+ gcc-c++ openssl openssl-devel pcre pcre-deve
```

1. 创建系统用户

```
#useradd fastdfs -M -s /sbin/nologin
#useradd nginx -M -s /sbin/nologin
```

2. 安装 FastFDS

```
#tar xf FastDFS_v5.01.tar.gz
#cd FastDFS
#./make.sh
#./make.sh install
```

3. 安装 Nginx

```
#cd ..
#tar xf fastdfs-nginx-module_v1.16.tar.gz
#tar xf nginx-1.7.0.tar.gz
#cd nginx-1.7.0

#./configure --user=nginx --group=nginx --prefix=/usr/local/nginx --add-module=
../fastdfs-nginx-module/src

#make
#make install
```

注意 在 Tracker Server 上安装 Nginx，不需要 --add-module=../fastdfs-nginx-module/src 这个配置项，而两个 Storage Server 的安装必须加上这个配置项。

7.1.3 Tracker Server 配置

1. 创建数据及日志存放目录

```
#mkdir -p /data/fastdfs/tracker
```

2. 修改 tracker.conf 配置

```
#vi /etc/fdfs/tracker.conf
```

更改下列两行配置：

```
base_path=/data/fastdfs/tracker
group_name=group1
```

3. 修改 nginx.conf 配置

```
#vi /usr/local/nginx/conf/nginx.conf
```

修改完成后如代码清单 7-1 所示，这是 Tracker Server 的一个负载均衡配置（代码内容可以从工程的 doc 目录 nginx.conf-tracker 文件中复制进来）。注意这里的 Tracker Server 开放访问端口为 84（非必须，只因为 80 端口已经作为其他用途）。

代码清单 7-1　Tracker Server 的 Nginx 配置

```
user  nginx nginx;
worker_processes 4;
pid /usr/local/nginx/nginx.pid;
worker_rlimit_nofile 51200;
events
{
use epoll;
worker_connections 20480;
}

http
{
    include       mime.types;
    default_type  application/octet-stream;
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request"'
            '$status $body_bytes_sent "$http_referer"'
            '"$http_user_agent" "$http_x_forwarded_for" "$request_time"';

    access_log /usr/local/nginx/logs/access.log main;

    upstream server_group1{
            server 192.168.1.215;
            server 192.168.1.216;
    }

    server {
            listen 84;
            server_name 192.168.1.214;
            location /group1 {
                include proxy.conf;
                proxy_pass http://server_group1;
            }
    }
}
```

4. 配置 Tracker Server 启动程序

使用如下指令配置 Tracker Server 的启动程序，并将其设置为随系统启动自动启动。

```
#cp /opt/FastDFS/init.d/fdfs_trackerd /etc/init.d/
#chkconfig --add fdfs_trackerd
#chkconfig fdfs_trackerd on
```

5. 配置 Nginx 的启动程序

使用下面指令创建一个启动文件：

```
#vi /etc/init.d/nginx
```

然后编辑（可以复制实例工程中 doc 目录的 nginx 文件内容）如代码清单 7-2 所示的内容。

代码清单 7-2 Nginx 启动程序

```
#!/bin/bash
# nginx Startup script for the Nginx HTTP Server
# it is v.0.0.2 version.
# chkconfig: - 85 15
# description: Nginx is a high-performance web and proxy server.
# It has a lot of features, but it's not for everyone.
# processname: nginx
# pidfile: /var/run/nginx.pid
# config: /usr/local/nginx/conf/nginx.conf
nginxd=/usr/local/nginx/sbin/nginx
nginx_config=/usr/local/nginx/conf/nginx.conf
nginx_pid=/var/run/nginx.pid
RETVAL=0
prog="nginx"
# Source function library.
. /etc/rc.d/init.d/functions
# Source networking configuration.
. /etc/sysconfig/network
# Check that networking is up.
[ ${NETWORKING} = "no" ] && exit 0
[ -x $nginxd ] || exit 0
# Start nginx daemons functions.
start() {
if [ -e $nginx_pid ];then
    echo "nginx already running...."
    exit 1
fi
    echo -n $"Starting $prog: "
    daemon $nginxd -c ${nginx_config}
    RETVAL=$?
    echo
    [ $RETVAL = 0 ] && touch /var/lock/subsys/nginx
    return $RETVAL
}
# Stop nginx daemons functions.
stop() {
        echo -n $"Stopping $prog: "
        killproc $nginxd
        RETVAL=$?
```

```
        echo
        [ $RETVAL = 0 ] && rm -f /var/lock/subsys/nginx /var/run/nginx.pid
}
# reload nginx service functions.
reload() {
    echo -n $"Reloading $prog: "
    #kill -HUP `cat ${nginx_pid}`
    killproc $nginxd -HUP
    RETVAL=$?
    echo
}
# See how we were called.
case "$1" in
start)
        start
        ;;
stop)
        stop
        ;;
reload)
        reload
        ;;
restart)
        stop
        start
        ;;
status)
        status $prog
        RETVAL=$?
        ;;
*)
        echo $"Usage: $prog {start|stop|restart|reload|status|help}"
        exit 1
esac
exit $RETVAL
```

将上面文件保存后，使用下列指令，修改为任何人可执行，并将其设定为随系统启动自动启动：

```
#chmod 755 /etc/init.d/nginx
#chkconfig --add nginx
#chkconfig nginx on
```

7.1.4　Storage Server 配置

两台 Storage Server 都要进行下列相关的配置。

1. 创建数据及日志保存目录

```
#mkdir -p /data/fastdfs/storage/data
```

2. 修改 storage.conf 配置

```
# vi /etc/fdfs/storage.conf
```

修改下列配置项，其他保持默认设置：

```
group_name=group1
base_path=/data/fastdfs
store_path0=/data/fastdfs/storage
tracker_server=192.168.1.214:22122
run_by_group=fastdfs
run_by_user=fastdfs
file_distribute_path_mode=1
rotate_error_log=true
```

3. 修改 mod_fastdfs.conf 配置

```
#cp /opt/fastdfs-nginx-module/src/mod_fastdfs.conf /etc/fdfs/
#vi /etc/fdfs/mod_fastdfs.conf
```

修改下列各项配置：

```
connect_timeout=30
tracker_server=192.168.1.214:22122
group_name=group1
url_have_group_name = true
store_path_count=1
store_path0=/data/fastdfs/storage
```

4. 修改 nginx.conf 配置

Storage Server 的 Nginx 配置如代码清单 7-3 所示，监听 80 端口，并使用 fastdfs-nginx-module 模块（代码内容保存在工程的 doc 目录 nginx.conf-storage 文件中）。

代码清单 7-3 Storage Server 的 Nginx 配置

```
user nginx nginx;
worker_processes 4;
pid /usr/local/nginx/logs/nginx.pid;
worker_rlimit_nofile 1024;

events {
    use epoll;
```

```
    worker_connections 1024;
}

http {

    include mime.types;
    server_names_hash_bucket_size 128;
    client_header_buffer_size 32k;
    large_client_header_buffers 4 32k;
    client_max_body_size 20m;
    limit_rate  1024k;

    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
    '$status $body_bytes_sent "$http_referer" '
    '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /usr/local/nginx/logs/access.log main;

    server {
        listen 80;
        server_name localhost;

        location /group1/M00{
            root /data/fastdfs/storage/data;
            ngx_fastdfs_module;
        }
    }
}
```

使用下列指令创建一个 M00 软连接，让配置中的 M00 同样指向 data 目录。

```
#ln -s /data/fastdfs/storage/data /data/fastdfs/storage/data/M00
```

5. 配置 Storage Server 的启动程序

```
#cp /opt/FastDFS/init.d/fdfs_storaged /etc/init.d/
#chkconfig --add fdfs_storaged
#chkconfig fdfs_storaged on
```

6. 配置 Nginx 的启动程序

Nginx 的启动程序配置与 Tracker Server 的配置相同。如果不使用自定义的启动程序，也可以使用下列指令启动 Nginx。

```
#/usr/local/nginx/sbin/nginx
```

7.1.5 启动服务

1. 启动 Tracker Server

```
#service fdfs_trackerd start
#service nginx start
```

2. 启动 Storage Server

```
#service fdfs_storaged start
#service nginx start
```

启动后，可以使用下列指令来查看各个服务的进程。

```
#ps -ef|grep fdfs
#ps-ef|grep nginx
```

如果能查看到服务进程，一般就说明已经启动成功。

7.1.6 客户端测试

1. 在 Tracker Server 中配置一个客户端

```
#vi /etc/fdfs/client.conf
```

修改下列配置项：

```
base_path=/data/fastdfs
tracker_server=192.168.10.214:22122
```

2. 查看服务的运行情况

在 Tracker Server 使用下列指令可以查看服务的运行情况：

```
#fdfs_monitor /etc/fdfs/client.conf
```

3. 测试文件上传

如果在当前路径（例如 /opt）中存在一个图片文件：01.jpg，即可使用下列指令来测试上传文件：

```
#fdfs_upload_file /etc/fdfs/client.conf 01.jpg
```

上传成功后将返回已经保存的文件标识，它包含组名、路径和文件名，如下所示：

```
group1/M00/00/00/wKgB2FdH892AclCqAAA2FfBeCgg517.jpg
```

4. 使用浏览器访问文件

使用上面配置的 Tracker Server 的 Web 服务端口，就可以使用下面完整的 URL 访问上面上传的文件：

```
http://192.168.1.214:84/group1/M00/00/00/wKgB2FdH892AclCqAAA2FfBeCgg517.jpg
```

注意　上面这个链接只是本地局域网的地址，如果要在互联网中使用，必须使用外网 IP 或者域名。

现在，就可以在安装的两台 Storage Server 服务器中其中一台的 /data 目录中找到已经保存的文件。

上面测试成功，表明 FastFDS 安装成功，并且已经正常运行，接着介绍如何在应用系统中使用分布式文件系统。

上面安装方法参考于 http://itindex.net/detail/49559-fastdfs-nginx-%E9%87%8F%E-7%BA%A7。

7.2　FastFDS 客户端

FastDFS 有 Java 的客户端 API，可以实现文件的上传、下载和删除等操作。在实例中，将使用一个更加简单的由第三方提供的开源 FastFDS_Client 组件，更加轻量地使用 FastFDS 分布式文件系统的功能。FastFDS_Client 是由 tobato 提供的专门为 Spring Boot 应用编写的 FastFDS 客户端应用。需要了解更多有关 FastFDS_Client 的细节，可以从下列 URL 中查看或下载它的源代码。

```
https://github.com/tobato/FastDFS_Client.git
```

本章实例工程由以下两个模块组成：

数据库管理模块：neo4j；

Web 应用模块：webapp。

其中，数据库管理模块使用 Neo4j 数据库提供数据存取的功能，Web 应用模块提供文件的上传和管理等操作。

7.2.1 客户端配置

首先，在实例工程的 Web 应用模块中的 Maven 依赖管理中引用 FastFDS_Client 的依赖配置，如代码清单 7-4 所示。在工程的主程序中增加一个注解：@Import (FdfsClientConfig.class)，以导入 FastFDS_Client 的配置。

代码清单 7-4 FastFDS_Client 依赖

```
<dependency>
        <groupId>com.github.tobato</groupId>
        <artifactId>fastdfs-client</artifactId>
        <version>1.25.1-RELEASE</version>
    </dependency>
```

然后，在 Web 应用模块的工程配置文件 application.yml 中增加如代码清单 7-5 所示的配置。其中，trackerList 是配置 Tracker Server 的列表，因为只安装了一个 Tracker Server，所以只要配置一个即可。

代码清单 7-5 FastFDS_Client 配置

```
fdfs:
    soTimeout: 1501
    connectTimeout: 601
    thumbImage:
        width: 150
        height: 150
    trackerList:
        - 192.168.1.214:22122
#     - 192.168.1.215:22122
spring.jmx.enabled: false
```

7.2.2 客户端服务类

为了能使用 FastFDS_Client，需要编写一个调用 FastFDS_Client 的服务类 Fastefs-Client，如代码清单 7-6 所示。其中文件上传时调用了 FastFDS_Client 的 uploadFile，文件删除时调用了 FastFDS_Client 的 deleteFile。

代码清单 7-6 使用 FastFDS 上传和删除文件

```
@Service
public class FastefsClient {
    @Autowired
    protected FastFileStorageClient storageClient;
```

```
    public String uploFile(MultipartFile file){
        String fileType = FilenameUtils.getExtension(file.getOriginalFilename
()).toLowerCase();
        StorePath path = null;
        try {
            path = storageClient.uploadFile(file.getInputStream(), file.get
Size(), fileType, null);
        }catch (IOException e){
            e.printStackTrace();
        }
        if(path != null)
            return path.getFullPath();
        else
            return null;
    }
    public void deleteFile(String fullPath){
        storageClient.deleteFile(fullPath);
    }
}
```

7.3　使用定制方式上传图片

所谓定制方式，就是设计一个图片选择框，可以使用调整大小和选择取图范围等手段设定上传的图片文件。

7.3.1　实体建模

为了演示文件上传，使用 Neo4j 数据库创建一个商品节点实体，如代码清单 7-7 所示。程序中省略了 Getter 和 Setter 方法，这些方法可以用 IDEA 编辑器自动生成，其中 picture 属性用来保存单个图片的链接地址，contents 属性用来保存在富文本编辑器中编辑的内容，其他属性如名称、简要说明、定价等结合起来主要体现一个商品的基本信息。

代码清单 7-7　商品节点实体建模

```
@NodeEntity
public class Goods {
    @GraphId
    private Long id;
    private String name;
    private String brief;
```

```
    private String picture;
    private String price;
    private String contents;
    @DateLong
    @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private Date create;
private Long shopid;
......
}
```

7.3.2 上传图片

1. 上传图片对话框设计

在编辑商品的过程中上传图片时，将打开一个上传图片对话框，对话框使用JavaScript设计，如代码清单7-8所示。在对话框中设计了三个按钮，分别是确定、删除和取消，其中确定按钮将调用sureChoose方法对图片进行裁剪，然后将上传图片的链接地址导入编辑商品的页面，由商品编辑的页面保存。“/pic/upload”是连接控制器的URL，使用这个链接将由控制器返回对话框的上传图片页面设计“uploa-pic.html”。

代码清单7-8 上传图片对话框设计

```
function picUp() {
    var picWidth = 720, picHeight = 400, callback = setImgUrl;
    var isSubmit = false;
    rt = pic.dialog.show({
        src: "/pic/upload",
        name: "_uploadPic_iframe",
        title: "上传图片",
        width: 750,
        height: 550,
        titleLineType: 'g-topbg',
        btn: {
            yes: "确定",
            del: "删除",
            no: "取消",
            no_style: 'white',
            del_style: 'orange',
            yes_before_close: function (win) {
                if (!isSubmit) {
                    isSubmit = true;
                    win.sureChoose(function (data) {
                        if (data) {
```

```
                        callback(data);
                        rt.hide();
                    } else {
                        isSubmit = false;
                    }
                });
            }
            return false;
        },
        del_before_close: function (win) {
            win.delChoose(function (data) {
                if (data) {
                    delImaConfirm(data);
                }
            });
        }
    },
    load: function (win) {
        win.page.upload.init(picWidth, picHeight);
    }
});
}
```

2. 上传图片页面设计

代码清单 7-9 是设计上传图片页面的部分 HTML 代码，其中通过一个类型为 file 的输入文本框，从本地中选择需要上传的图片文件，然后将图片展示在编辑界面上。在编辑界面上，使用了一个图片选择框，这样，不但可以调整上传图片的宽度和高度，还可以在图片中选择特定的区域进行裁剪。

代码清单 7-9　上传图片页面设计

```
<div class="img-view">
    <div class="up-tit">下图为您的图片展示，请注意是否清晰。</div>
    <div class="upload-box">
        单击上传
        <input id="pictureFile" name="pictureFile" type="file" class="file"
onchange="uploadPic_submit(this)"/>
    </div>
    <div class="view-box">
        <div class="view" id="view"><img/></div>
        <input type="hidden" value="" id="p1"/><!-- 左上坐标 -->
        <input type="hidden" value="" id="p2"/><!-- 右上坐标 -->
        <input type="hidden" value="" id="p3"/><!-- 右下坐标 -->
```

```
            <input type="hidden" value="" id="p4"/><!-- 左下坐标 -->
        </div>
        <div class="file-type">支持类型：jpg、jpeg、png。<span class="error">格式
不正确</span></div>
        <div class="reupload">
            <div class="l">
                <div class="newUpDiv"><p>选择文件</p>
                    <input id="changeFile" class="g-btn g-btn-white"name="pictureFile"
type="file" onchange="uploadPic_submit(this)"/>
                </div>

            </div>
            <div class="clear"></div>
        </div>
    </div>
```

在上传图片的对话框中单击上传文件后，将调用 ajaxFileUpload 方法，如代码清单 7-10 所示。其中的链接地址“/pic/uploadPic”将调用文件上传控制器，实现文件上传功能。

代码清单 7-10 上传文件的 Ajax 方法定义

```
function ajaxFileUpload(id){
    var url = '/pic/uploadPic';
    $.ajaxFileUpload({
        url : url,                    //需要链接到服务器地址
        fileElementId : id,           //文件选择框的 id 属性
        dataType : 'json',            //服务器返回的格式，可以是 json
        success : function(data) {
            if(data.errorMsg){
                showMsg(data.errorMsg, "错误");
            }else{
                page.upload.finish(data.pathInfo,data.width,data.height);
            }
        }
    });
}
```

3. 上传图片控制器设计

上传图片控制器的设计，将调用 7.2.2 节定义的 FastefsClient 服务类，直接与 FastFDS 服务器打交道，如代码清单 7-11 所示。这里必须注意区别以下两种类型的文件

路径：

filename：这是 FastefsClient 使用的与 FastFDS 服务器进行交互通信时使用的文件路径，它由组名、路径和文件名组成。

pathHead + filename：这是可以在浏览器的页面上使用的完整图片文件路径，pathHead 由 Tracker Server 的域名或 IP 地址及其端口组成。

将在数据库中保存 filename 和 pathHead 这两个属性，在页面视图中调用的返回参数中，使用了 pathInfo（pathHead+filename）参数返回文件的完整路径。

代码清单 7-11　上传图片控制器设计

```
    @Value("${file.path.head:http://192.168.1.214:84/}")
    private String pathHead;
    @Autowired
    private FastefsClient fastefsClient;
    @RequestMapping(value = "/uploadPic", method = RequestMethod.POST)
        public void uploadPic(@RequestParam("pictureFile") MultipartFile multipar
tFile,HttpServletRequest request,HttpServletResponse response) {
            try {
                String filename = fastefsClient.uploFile(multipartFile);
                Long shopid = 1L;
                ......
                BufferedImage image = ImageIO.read(multipartFile.getInputStream());

                Map<String, Object> data = new HashMap<String, Object>();
                data.put("pathInfo", pathHead+filename);
                data.put("width", image.getWidth());
                data.put("height", image.getHeight());

                ObjectMapper mapper = new ObjectMapper();
                String ret =  mapper.writeValueAsString(data);

                response.setContentType("text/html;charset=utf8");
                response.getOutputStream().write(ret.getBytes());
                response.flushBuffer();
            }catch (IOException e){
                e.printStackTrace();
            }

        }
```

4. 上传图片效果图

上面设计最终的运行效果如图 7-2 所示。因为使用了定制的方式，也就是使用图片

选择框重新选择和调整上传的图片，所以这个过程完成后，后台会对已经上传的文件进行裁剪，裁剪后删除原来的旧文件，并保存裁剪下来的新文件。

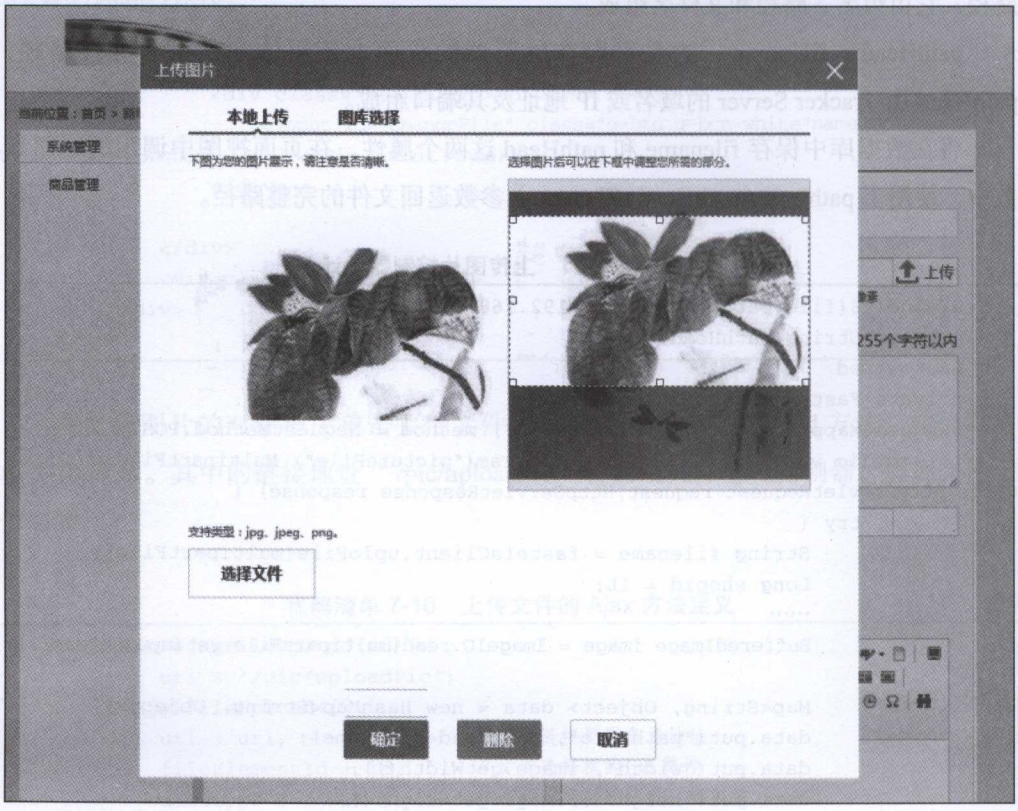


图 7-2 使用定制方式上传图片的效果图

7.4 使用富文本编辑器上传图片

在实际应用中，常常需要使用富文本编辑器来编辑一些比较复杂的内容，例如在实例中，对商品内容的描述可能既有文字说明，又有图片展示。在富文本编辑器中上传图片，具体取决于编辑器本身的设计。当然，一般富文本编辑器的设计都有一些可供配置的选项，例如，Ueditor 就是一个完全开源的可以定制的富文本编辑器。

7.4.1 使用富文本编辑器

使用 Ueditor 的页面，必须加入一些 JS 的引用和配置。因为在商品新建和编辑页面

中都需要使用 Ueditor，所以都必须加入对它的引用，如代码清单 7-12 所示。

代码清单 7-12　使用 ueditor 的引用

```
    <script type="text/javascript" charset="utf-8">
        window.UEDITOR_HOME_URL = "/ueditor/";
    </script>
    <script type="text/javascript" charset="utf-8" th:src="@{/ueditor/editor_
config.js}"></script>
    <script type="text/javascript" charset="utf-8" th:src="@{/ueditor/editor
_all.js}"></script>
```

然后在 Ueditor 的配置文件 editor_config.js 中修改上传图片的提交地址、路径等参数，将提交地址指向编写的控制器提供的链接地址，代码如下：

```
,imageUrl:"/pic/uploadimg"          // 图片上传提交地址
,imagePath:""                       // 图片修正地址
,imageFieldName:"upfile"            // 图片数据的 key，需要在后台修改对应文件的对应参数
```

7.4.2　实现文件上传

代码清单 7-13 是实现使用富文本编辑器的文件上传的控制器设计。这个控制器的设计与代码清单 7-11 控制器的设计差不多，只是返回的参数略有不同，以适合调用者——Ueditor 的使用，其中返回参数中的 url 就是一个完整的图片路径。

代码清单 7-13　Ueditor 图片上传控制器设计

```
    //ueditor 图片上传
    @RequestMapping(value = "/uploadimg", method=RequestMethod.POST, produces=
"text/html;charset=UTF-8")
    public void uploadimg(@RequestParam("upfile") MultipartFile upfile,Http
ServletRequest request,HttpServletResponse response){
        try {
            String filename = fastefsClient.uploFile(upfile);

            Long shopid = 1L;

            Map<String, Object> data = new HashMap<String, Object>();
            data.put("original", upfile.getOriginalFilename());
            data.put("url", pathHead+filename);
            data.put("title", "");
            data.put("state", "SUCCESS");

            ObjectMapper mapper = new ObjectMapper();
            String ret =  mapper.writeValueAsString(data);
```

```
            response.setContentType("text/html;charset=utf8");
            response.getOutputStream().write(ret.getBytes());
            response.flushBuffer();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
```

使用 Ueditor 上传图片的效果如图 7-3 所示。图片上传后，选择图片，单击富文本编辑器中的上传图片按钮，可以查看上传图片的简要信息。

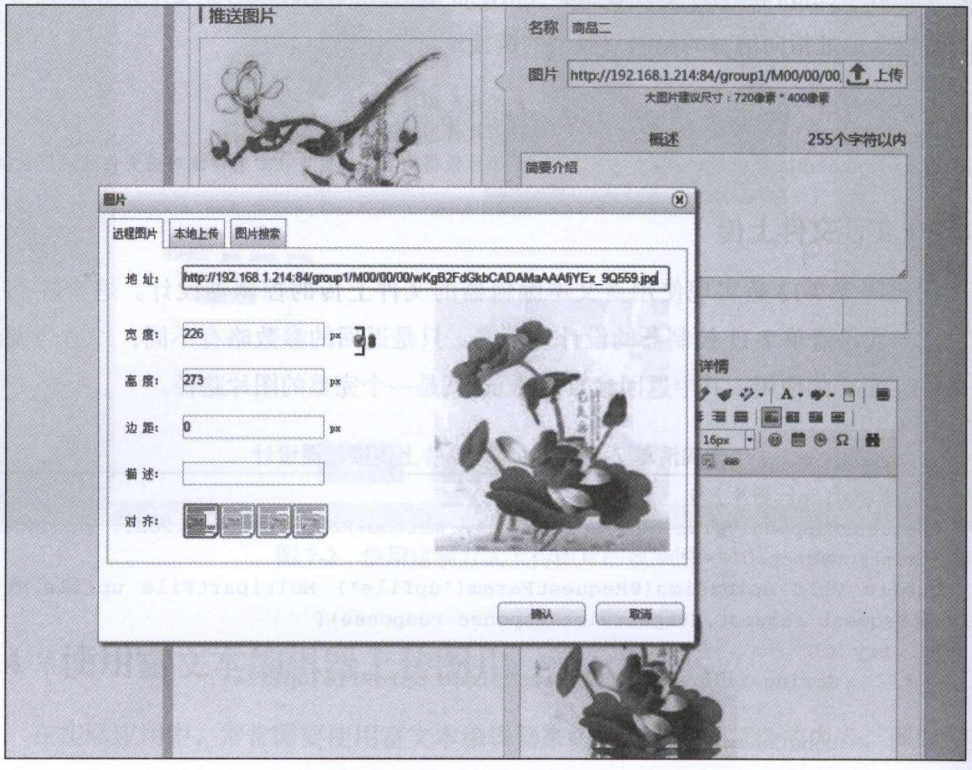


图 7-3 使用富文本编辑器上传图片效果图

7.5 使用本地文件库

对于已经上传的图片，可以创建一个文件库来管理，这样不但可以重复利用图片，

还可以编辑，如果不再需要了，可以执行删除操作。

7.5.1　本地文件库建模

在 Neo4j 数据库中增加一个用来保存图片信息的图片节点，代码清单 7-14 是图片节点 Picture 的实体建模，其中 fileName 用来保存从 FastFDS 返回的文件路径，pathInfo 用来保存完整的文件链接前半部分路径，即代码清单 7-11 中的 pathHead 参数的值。

代码清单 7-14　图片节点实体建模

```
@NodeEntity
public class Picture {
    @GraphId
    private Long id;
    private String pathInfo;
    private String fileName;
    private int width;
    private int height;
    private String flag;
    @DateLong
    private Date create;
private Long shopid;
......
}
```

7.5.2　文件保存方法

代码清单 7-15 是一个保存文件的后台线程，使用后台线程来执行保存文件的方法，将会在不影响界面上操作的情况下，从后台中稍稍地执行 savePic 方法。

代码清单 7-15　保存文件的线程

```
AsyncThreadPool.getInstance().execute(new Runnable() {
            @Override
            public void run() {
                try {
                    savePic(upfile, filename ,shopid);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
```

savePic 方法的定义如代码清单 7-16 所示，它调用数据库服务类 PictureService，将

文件信息保存到数据库中。其中使用 ImageIO 来读取文件的高度和宽度。

代码清单 7-16 保存文件的方法

```
    @Autowired
        private PictureService pictureService;
                private void savePic(MultipartFile multipartFile, String filename,
Longshopid) throws Exception{
            BufferedImage image = ImageIO.read(multipartFile.getInputStream());

            Picture picture = new Picture();
            picture.setFileName(filename);
            picture.setHeight(image.getHeight());
            picture.setWidth(image.getWidth());
            picture.setPathInfo(pathHead);
            picture.setCreate(new Date());
            picture.setShopid(shopid);

            pictureService.create(picture);
        }
```

数据库服务类 PictureService 的定义如代码清单 7-17 所示，它调用资源库接口 pictureRepository 和分页查询服务类 pagesService，提供增删查改及其分页查询功能。

代码清单 7-17 数据库服务类 PictureService

```
@Service
public class PictureService {
    @Autowired
    private PictureRepository pictureRepository;
    @Autowired
    private PagesService<Picture> pagesService;

    public Picture findById(Long id) {
        return pictureRepository.findOne(id);
    }

    public Picture create(Picture picture) {
        return pictureRepository.save(picture);
    }

    public Picture update(Picture picture) {
        return pictureRepository.save(picture);
    }

    public void delete(Long id) {
```

```
        Picture picture = pictureRepository.findOne(id);
        pictureRepository.delete(picture);
    }

    public Picture findByName(String name){
        return pictureRepository.findByFileName(name);
    }

    public void delete(Picture picture){
        pictureRepository.delete(picture);
    }

    public Iterable<Picture> findAll(){
        return pictureRepository.findAll();
    }

    public Page<Picture> findPage(PictureQo pictureQo){
        Pageable pageable = new PageRequest(pictureQo.getPage(), pictureQo.
getSize(), new Sort(Sort.Direction.ASC, "id"));

        Filters filters = new Filters();
        if (!StringUtils.isEmpty(pictureQo.getFileName())) {
            Filter filter = new Filter("name", pictureQo.getFileName());
            filters.add(filter);
        }
        if (!StringUtils.isEmpty(pictureQo.getShopid())) {
            Filter filter = new Filter("shopid", pictureQo.getShopid());
            filter.setComparisonOperator(ComparisonOperator.EQUALS);
            filter.setBooleanOperator(BooleanOperator.AND);
            filters.add(filter);
        }

        return pagesService.findAll(Picture.class, pageable, filters);
    }
}
```

7.5.3　文件库管理

保存图片信息之后，可以取出已经上传的图片列表，按需选择可用的图片，不需要的图片，也可以删除。需要注意的是，删除图片时，不但要删除文件库 Picture 的图片信息，而且要删除 FastFDS 服务器上的文件。

代码清单 7-18 是取图片列表的控制器设计，它将返回一个 Page 对象，其中包含 Picture 列表，列表将由视图处理并按页显示出来以供用户选择使用。

代码清单 7-18 取图片列表控制器设计

```
        @RequestMapping(value = "/listPic", method = RequestMethod.POST)
    @ResponseBody
    public Page<Picture> listPic(PictureQo pictureQo) throws IOException{
        Long shopid = 1L;
        pictureQo.setShopid(shopid);
        return pictureService.findPage(pictureQo);
    }
```

代码清单 7-19 是取图片列表的视图设计部分的 js 编码，它从上面定义的控制器中取得列表后，按每行 4 张图片、每页 2 行的格式来显示列表。

代码清单 7-19 取图片列表视图设计部分的 js 编码

```
//分页数据
function getDataHtml(pageNo,pagesize) {
    $.ajax({
        url: "/pic/listPic",
        dataType: "json",
        type: "POST",
        cache: false,
        data: {page: pageNo-1,size: pagesize || 8},
        success: function (data) {
            var $list = $('#upload-list').empty();
            $.each(data.content, function (i, v) {
                var html = "";
                html += '<div class="upload-item">'+
                    '<div class="img"><img src="'+ v.pathInfo + v.fileName + '"
/></div>'+
                    '<p>'+v.width+'x'+ v.height+'</p>'+
                    '<div class="selected"></div>'+
                    '</div>';

                $list.append($(html));
            })
            page.photos.setPosition();
            document.getElementById('pagebar').innerHTML = PageBarNumList.get
PageBar(data.number+1, data.totalPages, 3, 'getDataHtml',pagesize || 8,true);
        },
        error: function (e) {}
    });
}
```

最后完成的效果如图 7-4 所示。这样，当在图片中单击选择一张图片后，再单击确定按钮即可使用这张图片，单击删除按钮将删除这张图片。

图 7-4　本地文件库管理效果图

7.6　运行与发布

本章实例工程的完整代码可以在 IDEA 中直接从 GitHub 中检出：https://github.com/chenfromsz/spring-boot-files.git。

检出工程后，在 webapp 模块的配置文件 application.yml 中按照各种服务器的安装情况配置连接 Neo4j 数据库的 URL、用户名和密码，配置连接 FastFDS 分布式文件系统的 Tracker Server 的 trackerList 列表地址和端口，配置 file.path.head 的 Tracker Server 浏览器地址和端口，如代码清单 7-20 所示。

代码清单 7-20　webapp 模块配置

```
server:
    port: 80

neo4j:
    datasource:
        url: http://192.168.1.221:7474
```

```
        username: neo4j
        password: 12345678

fdfs:
    soTimeout: 1501
    connectTimeout: 601
    thumbImage:
        width: 150
        height: 150
    trackerList:
        - 192.168.1.214:22122
#     - 192.168.1.215:22122
spring.jmx.enabled: false

file.path.head: http://192.168.1.214:84/
```

然后在IDEA的Edit Configuration中增加一个Spring Boot启动配置，模块选择webapp，工作目录选择模块所在的根目录，主程序选择com.test.webapp.WebappApplication，并将配置保存为webapp。

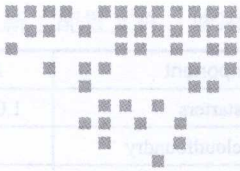
运行webapp配置项目，应用启动成功后，打开网址http://localhost，即可体验使用分布式文件系统的功能。

如果需要发布工程，可以在IDEA的Edit Configuration中增加一个Maven配置，工作目录选择工程根目录，在命令行输入clean package，并保存配置为mvn。然后运行配置项目mvn，即可执行打包操作。

7.7 小结

本章使用FastFDS安装一个分布式文件系统，使用FastFDS_Client演示使用定制方式和富文本编辑器的方式上传图片文件的功能，并建立一个本地文件库来管理上传文件。通过本章实例的演示，我们发现，在Spring Boot框架中使用FastFDS也是非常容易的。要提高一个大型分布式系统的图片文件访问性能和设计一个可以动态扩容的文件系统，使用分布式文件系统是最佳的选择。

使用分布式文件系统，可以提升服务系统在存取文件方面的性能，但对于提升整个服务体系的性能来说，这还远远不够，下一章将介绍使用云应用开发工具，开发高可用的微服务，以提升服务系统整体性能。



第 8 章 Chapter 8

云应用开发

2014 年 6 月，Pivotal 团队正式发布了 Spring Cloud 1.0.0 版本。Spring Cloud 是一套云应用开发工具集，为分布式的微服务开发提供了一整套简单易用的使用工具。Spring Cloud 主要包含配置管理、服务发现、动态路由、负载均衡、断路器、安全管理、事件总线、分布式消息等组件的开发工具包。

表 8-1 列出了 Spring Cloud 组件和对应的版本，在第 5 章的安全管理和第 6 章的 SSO 设计中使用了其中的 spring-cloud-secutiry。Spring Cloud 在 Brixton.M4 版本之后，Maven 可以使用 spring-cloud-starter-parent 方式来简化工程的依赖配置，我们使用的 Spring Cloud 版本是 Brixton.M5，对应的 Spring Cloud 版本是 1.1.0，Spring Boot 的版本是 1.3.2。

表 8-1 Spring Cloud 组件列表

Component	Angel.SR6	Brixton.RELEASE	Brixton.BUILD-SNAPSHOT
spring-cloud-aws	1.0.4.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-bus	1.0.3.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-cli	1.0.6.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-commons	1.0.5.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-config	1.0.4.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-netflix	1.0.7.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-security	1.0.3.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT

（续）

Component	Angel.SR6	Brixton.RELEASE	Brixton.BUILD-SNAPSHOT
spring-cloud-starters	1.0.6.RELEASE		
spring-cloud-cloudfoundry		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-cluster		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-consul		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-sleuth		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-stream		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-zookeeper		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-boot	1.2.8.RELEASE	1.3.5.RELEASE	1.3.5.RELEASE
spring-cloud-stream-app-starters*			1.0.0.BUILD-SNAPSHOT
spring-cloud-task*			1.0.0.BUILD-SNAPSHOT

（上面资料来源：http://projects.spring.io/spring-cloud/#quick-start）

Spring Cloud 与 Spring Boot 关系密切，能够臻于完美的结合使用。

本章的实例工程使用了模块化设计，用来介绍 Spring Cloud 工具集中几个主要组件的具体使用，如表 8-2 所示。

表 8-2 实例工程模块

名称	项目	类型	功能
配置服务	config	Web 应用	为客户端提供配置管理及更新服务
发现服务	discovery	Web 应用	为客户端提供注册与发现服务等功能
监控服务	hystrix	Web 应用	为分布式服务提供监控管理等功能
数据服务	data	Web 应用	Neo4j 数据管理服务
Web 服务	web	Web 应用	Web 客户端应用

8.1 使用配置管理

一个项目工程总是需要一些配置，例如，要配置服务器的端口、访问数据库的参数或其他一些项目需要的参数等。而一个大型的分布式系统可能存在很多这样需要配置的项目工程，这时，配置管理就将是一个庞大的工程，所以弄不好很容易出错。因此对于一个分布式系统来说，迫切需要一个单独的系统来专门管理各个项目的配置。Spring Cloud 的配置管理就是这样的一个工具包。

使用 Spring Cloud 的配置管理开发工具包，只要创建一个简单的工程，就可以实现

分布式配置管理服务，它还支持在线更新，即如果更新了配置文件，使用这个配置文件的客户端不用重启就可以使用最新的配置。

8.1.1　创建配置管理服务器

为了给客户端提供配置管理的服务，要创建一个工程，用来构建一个配置管理服务器。首先，在工程的 Maven 管理中引用如代码清单 8-1 所示的依赖配置。

代码清单 8-1　配置管理服务器依赖配置

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
```

然后，创建一个主程序，如代码清单 8-2 所示。这是这个工程中我们创建的唯一一个类，其中注解 @EnableConfigServe 启用了配置管理服务的功能，注解 @EnableDiscoveryClient 启用了发现服务的客户端功能，表明这是使用发现服务的一个客户端。

代码清单 8-2　配置管理服务器主程序

```
@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class ConfigApplication{
    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }
}
```

这样，就创建了一个配置管理服务器。它能为客户端提供配置文件的管理和更新等服务。

配置管理服务器中使用的文件格式与工程的本地配置文件格式一样，既可以使用“.properties”扩展名，也可以使用“.yml”扩展名。配置文件的存储目前支持使用本地存储、Git 以及 Subversion 等方式。在实例中，将使用 Git 的方式来存取配置文件，这需要在 Git 服务器上创建一个资源库，并将配置文件上传到创建的资源库中。为了方便演示配置管理服务的功能，将配置文件存放在 GitHub 中。如果是实际应用，建议使用自主创建的 Git 服务器。

在配置管理服务器的本地工程配置文件中，进行如代码清单 8-3 所示的设置，其中

“uri”指定资源库的位置。

代码清单 8-3 文件资源库配置

```
spring:
    cloud:
        config:
            server:
                git:
                    uri: https://github.com/chenfromsz/spring-cloud-config
-repo
```

该资源库包含如下文件，并且为了演示目的，文件中有一些简单的配置内容。要查看配置文件的内容，可以在浏览器中输入如代码清单 8-3 所示的 uri。关于如何使用这些配置文件，将在下一小节中介绍。

```
Application.yml
web.yml
web-development.yml
data.yml
data-development.yml
```

8.1.2 使用配置管理的客户端

客户端要使用配置管理服务，首先要在工程中的 Maven 依赖管理中加入配置管理组件 spring-cloud-starter-config 的依赖，如代码清单 8-4 所示。使用配置管理服务之后，如果本地的配置文件与配置管理服务器的配置文件有相同的配置项，将优先使用配置管理服务器的配置项，也就是说本地的配置项将会被覆盖。

代码清单 8-4 使用配置管理的客户端依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

其次，使用配置管理的客户端必须在本地的配置中，事先做好连接配置管理服务器，以及需要使用由配置管理服务器提供的配置文件等参数的设定。

各个使用配置管理的客户端项目中一个名为 bookstrap.yml 的本地配置文件，就是用来设定连接配置管理服务器、应用的名称，以及需要由配置管理服务器提供的配置文件等参数的，如代码清单 8-5 所示。配置中的一些参数解释如下：

uri：设定连接配置管理服务器的地址和端口。

name：用来指定应用的名称和配置文件的名称。

profiles：可以理解为使用配置文件名称的后缀部分。例如这个配置本身，因为 profiles 设定了 development，所以它使用的配置文件将是 web-development.yml 或 web-development.properties。如果不使用 profiles 这个参数，即使用没有文件名后缀部分的配置文件，例如，这个配置将使用 web.yml 或 web.properties 配置文件。

如果在资源库中有 application.yml 或 application.properties 文件，则默认加载。

另外，也可以在 config 参数下面使用 name 和 profiles 来指定配置文件的名称和后缀，即与上面的应用名称分开进行配置。这里使用上面的配置方法。

使用 name 和 profiles 这两个参数，可以设定不同环境使用不同的配置文件，这对于在开发环境中和在生产环境中使用不同的配置文件来说，是很方便的，只要更改 profiles 参数即可。

代码清单 8-5　客户端使用配置管理的设置

```
spring:
    application:
        name: web
    profiles:
        active: development
    cloud:
        config:
            uri: http://localhost:8888
```

设定这些配置参数之后，就可以使用配置管理服务器提供的服务了。可以像下面的程序片段那样，在需要用到配置的地方引用配置文件中的配置项，这个引用假设配置文件中包含 cloud.sample.msg 这个配置。

```
@Value("${cloud.sample.msg}") String msg;
String configValue = environment.getProperty("cloud.sample.msg", "undefined");
```


注意　对于客户端中的一些启动参数的配置，如应用服务的端口设定、数据库的连接地址、用户和密码等配置是在工程启动时加载，并且中途不能变更，所以这些启动参数虽然可以使用配置管理的配置，但不能使用自动更新功能，这应该是可以理解的。

测试上述一些配置的功能，首先启动实例工程的 config 模块，然后启动 web 模块，这时在它们的控制台上都可以看到加载了下列配置文件：

```
web-development.yml
web.yml
application.yml
```

虽然 web.yml 也被加载了，但是因为只是调用了 web-development.yml，所以只有这个文件的配置内容才会被读取。可以使用一个测试程序来验证一下，如代码清单 8-6 所示，我们设计了一个控制器。控制器提供了一个链接地址“/test”，当访问这个地址时，将从配置文件中读取配置参数“cloud.sample.msg”的值，并做出相应输出，如果读取失败，将取默认值 World。

代码清单 8-6 使用配置管理的测试程序 1

```
@RestController
public class TestController {
    @Value("${cloud.sample.msg:World}") String msg;

    @RequestMapping("/test")
    String test() {
        return "Hello " + msg + "!";
    }
}
```

在浏览器上输入网址 http://localhost:9001/test，因为在同一台机器上演示，web 项目使用了 9001 的端口。

假如在配置文件 web-development.yml 或 application.yml 中包含“cloud.sample.msg”配置项，就会输出它的值，否则输出：Hello World！现在 web-development.yml 包含如下的内容：

```
cloud:
    sample:
        msg: web-development
```

即上面浏览器正确的输出结果应该是如下的结果，这正是我们期望得到的输出结果。

```
Hello web-development!
```

如果暂时不需要使用配置管理服务器提供的配置，只想使用本地的配置文件，可以只把代码清单 8-4 中的依赖注释掉即可。

8.1.3　实现在线更新

上一小节对配置管理服务器的配置文件的读取，只有在应用启动时才会加载。这显然是不够理想的，能不能在应用不重启的情况下，在线更新配置呢？当然是可以的。要实现在线更新，必须在需要使用更新配置的 Bean 中增加一个注解：@RefreshScope，即将代码清单 8-6 改成代码清单 8-7。

代码清单 8-7　使用配置管理的测试程序 2

```
@RestController
@RefreshScope
public class TestController {
    @Value("${cloud.sample.msg:World}") String msg;

    @RequestMapping("/test")
    String test() {
        return "Hello " + msg + "!";
    }
}
```

通过这样更改之后，再来测试一下。启动实例工程的 web 模块之后，再将上面配置文件的内容修改成如下，并提交到 Git 服务器上。

```
cloud:
    sample:
        msg: web-develop
```

现在刷新上面的浏览器还不能读取到最新的配置，因为在线更新还必须使用一个指令来触发。使用下列的指令才能触发更新，使用这个指令之后，可以在 web 模块的控制台上看到重新加载了配置文件。这时候再刷新浏览器，就可以看到输出了我们期望的结果：Hello web-develop!

```
curl -X POST http://localhost:9001/refresh
```

curl 是 UNIX 系统的指令，上面指令需要使用 POST 方式传送，执行上面指令需要有 Linux 类型的系统。如果你的机器是 Windows 系统，也不用担心，如果已经安装了 Git 客户端，打开 Git Bash 窗口，同样也能使用 UNIX 指令。注意上面指令参数的大小写敏感。

8.1.4 更新所有客户端的配置

使用上面的方法，如果有很多应用，就需要一个一个地触发更新配置，相当麻烦。有没有一种方法，可以更新所有使用配置管理的客户端配置呢？使用事件总线 Spring-cloud-bus，就可以在线更新所有连接配置管理服务器的客户端。这样，仅仅使用一条指令就可以更新所有客户端的配置。

因为 spring-cloud-bus 是使用分布式消息发布机制，通过 RabbitMQ 使用消息分发的方法来执行更新的。所以还必须安装 RabbitMQ 服务器，才能实现消息分发。RabbitMQ 服务器的安装可以参照附录 D。

要启用 spring-cloud-bus 的功能，首先在配置管理服务器和所有使用配置管理的客户端的 Maven 依赖管理中，都要增加如代码清单 8-8 所示的依赖配置。

代码清单 8-8 spring-cloud-bus 依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

其次，在配置管理服务器和所有连接配置管理服务器的客户端的本地配置文件中，增加如代码清单 8-9 所示的配置，即设定连接 RabbitMQ 服务器的 IP、端口、用户名和密码等参数。

代码清单 8-9 连接 RabbitMQ 服务器的配置

```
spring:
    rabbitmq:
        addresses: amqp://192.168.1.214:5672
        username: alan
        password: alan
```

上面配置假设你已经安装了 RabbitMQ 服务器，安装的 IP 地址为 192.168.1.214，使用默认端口为 5672，增加了一个用户，用户名和密码都为 alan，并且为用户赋予了可以使用消息服务的权限。

通过网址 http://192.168.1.214:15672 使用浏览器打开 RabbitMQ 服务器，并使用管理员用户登录进去，就可以打开 RabbitMQ 服务器的控制台，在这里可以查看连接的客户端和通道等信息，如图 8-1 所示。

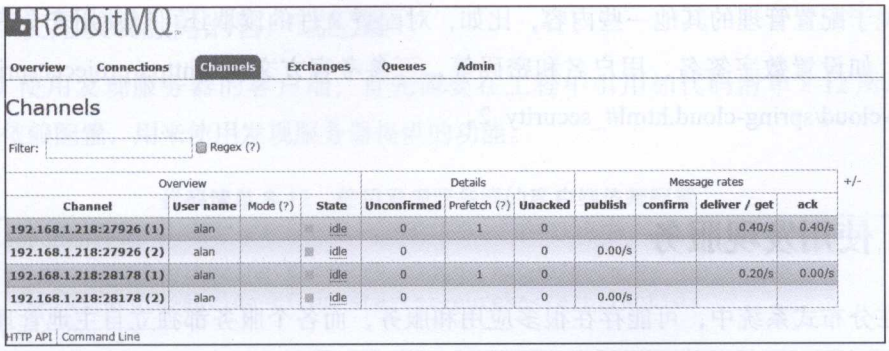


图 8-1　RabbitMQ 服务器连接通道情况

现在可以使用下列指令，更新所有在线的使用配置管理的客户端了，假设配置管理服务器的端口为 8888。

```
curl -X POST http://localhost:8888/bus/refresh
```

执行指令后，可以在配置管理服务器的控制台中看到它重新加载了一些客户端的配置文件。在连接配置管理服务器的客户端的控制台中同样可以看到重新加载了配置文件。

在本章的实例工程中，可以使用 web 和 data 两个模块来测试这个更新指令。执行指令后，可以看到两个模块的配置文件都被重新加载了，不管配置文件有没有更改。这样其实也就可以证明客户端的配置已经被更新了。如果你还不放心，也可以使用上一小节的测试程序来验证一下，两个模块都有相同的测试程序，即在浏览器中使用“/test”链接来测试。

上面这个指令跟上一小节的更新指令有点不同，主要体现在两点：第一，它必须使用配置管理服务器的地址来执行，第二，它的 URL 中多了一个“bus”。

但是，有时我们并不想一下子就更新所有的客户端，可能只需要更新一个客户端，这时既可以使用上一小节的更新指令，也可以在配置管理服务器中使用指定目标客户端来更新一个客户端的配置。例如，下面的指令指定更新名称为 web 的应用的所有更新，它使用了 destination 参数。

```
curl -X POST http://localhost:8888/bus/refresh?destination=web:**
```

上面所有的在线更新都必须要求需要使用更新的 Bean 中包含 @RefreshScope 注解的才能适用，这一点是毋庸置疑的。

关于配置管理的其他一些内容，比如，对配置文件的读取还可以使用安全措施和策略，如设置数字签名、用户名和密码等，可参考官方文档：http://projects.spring.io/spring-cloud/spring-cloud.html#_security_2。

8.2 使用发现服务

在分布式系统中，可能存在很多应用和服务，而各个服务都独立自主地管理自身的数据。在服务与服务之间，有时候可能需要互相共享一些数据，传统的做法是使用WebService，或者 SOAP 的方式，由服务提供者一方对外暴露接口，然后由服务消费者一方对接口进行访问，从而达到数据共享的目的。但是不管使用上面哪种方式，开发者都必须编写一些接口程序，可能还需要使用复杂的配置来实现。而使用 Spring Cloud，通过发现服务来实现服务之间的数据共享是轻而易举的。

8.2.1 创建发现服务器

要使用发现服务的功能，需要创建一个工程，用来构建一个发现服务器，在工程中需要引用如代码清单 8-10 所示的 Maven 依赖配置。

代码清单 8-10 发现服务器的依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

然后创建一个简单主程序，并在主程序中增加注解 @EnableEurekaServer，即启用发现服务器的功能，如代码清单 8-11 所示。

代码清单 8-11 发现服务器主程序

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApplication.class, args);
    }
}
```

8.2.2　使用发现服务的客户端配置

1）使用发现服务器的客户端，首先需要在工程中引用如代码清单 8-12 所示的 Maven 依赖配置，用来使用发现服务器提供的功能。

代码清单 8-12　使用发现服务器的客户端依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

2）在工程的主程序中，增加一个注解 @EnableDiscoveryClient，以启用发现服务的客户端功能。

3）在工程的配置文件中配置发现服务器的地址和端口，如代码清单 8-13 所示。假设发现服务器在本地运行，并且端口为 8761。

代码清单 8-13　使用发现服务的客户端配置

```
eureka:
    client:
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/
    instance:
        preferIpAddress: true
```

4）在 bookstrap.yml 配置文件中，配置应用的名称（应用名称的配置与使用配置管理服务器的配置相同，可以参考代码清单 8-5），代码如下。这个名称就是一个应用在发现服务器中的一个唯一标识，必须保证它的唯一性。

```
spring:
    application:
        name: web
```

8.2.3　发现服务器测试

要测试发现服务器，首先启动实例工程中的 discovery 项目，然后启动 config、data、web 等项目，如果各个服务器和客户端都运行了，在浏览器中打开地址 http://localhost:8761，即可打开发现服务器的控制台。

现在就可以看到如图 8-2 所示的情况，这里可以看到已经注册的 CONFIG、DATA、WEB 三个服务，分别对应实例工程的配置服务、数据服务、Web 服务三个模块的实例。

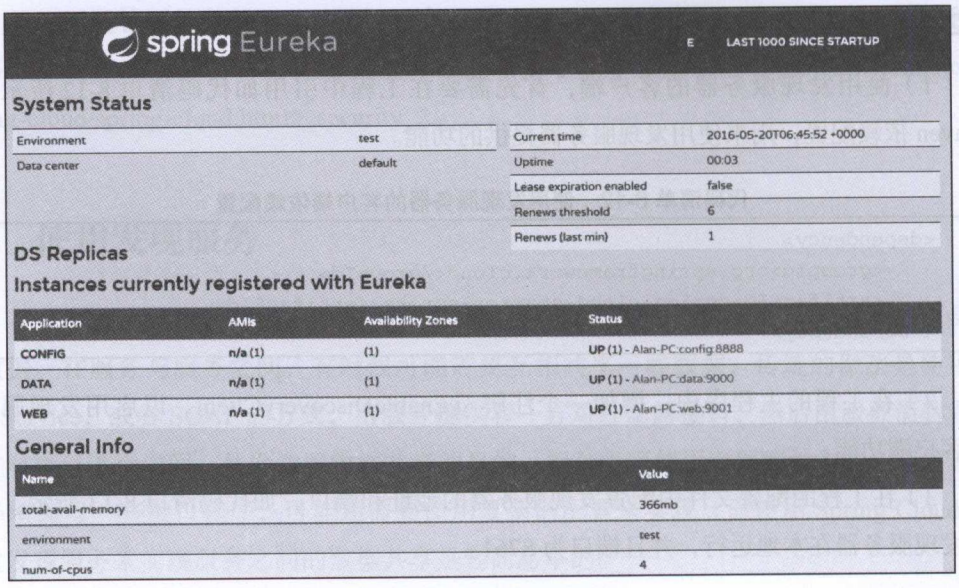


图 8-2 发现服务器控制台

8.3 使用动态路由和断路器

8.2 节创建了一个发现服务器，并向其注册了几个微服务（客户端），这一节将介绍如何在这些服务之间，使用动态路由、断路器和故障容错等功能。

当客户端发现服务器注册之后，客户端之间就可以通过 Zuul 路由代理协议，使用应用的名称来访问各自的 REST 资源。

8.3.1 依赖配置

要启用动态路由、断路器和服务监控等功能，首先需要在各个客户端的工程中引用如代码清单 8-14 所示的 Maven 依赖配置。

代码清单 8-14 启用路由和断路器等的依赖配置

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
        <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

其次在各个使用上述功能的工程的主程序中，增加如下注解，即可启用路由代理服务和启用监控服务功能。

```
@EnableZuulProxy
@EnableHystrix
```

8.3.2　共享 REST 资源

在实例工程的 data 模块，使用 Neo4j 图形数据库，创建了三个节点，分别是单位、用户、角色，并创建了两个关系，即用户隶属于单位的关系和用户拥有角色的关系。有关节点和关系的实体建模如表 8-3 所示。

表 8-3　实体建模列表

名称	实体	类型	备注
单位	Unit	节点	
用户	User	节点	
角色	Role	节点	
隶属关系	Belong	关系	开始节点：单位 结束节点：用户
拥有关系	Owner	关系	开始节点：用户 结束节点：角色

节点和关系实体的持久化，可以使用资源库的方式实现。代码清单 8-15 是用户节点的持久化实现，并且在实现持久化的同时也共享了 REST 资源。其中注解 @RepositoryRestResource 标注这个接口不但是一个数据库的资源库，也是一个 REST 资源共享，并把资源的名称和路径定义为 users。将在下一小节中介绍如何使用这个 REST 的资源共享。

代码清单 8-15　用户节点的 REST 资源库

```
@RepositoryRestResource(collectionResourceRel = "users", path = "users")
public interface UserRepository extends GraphRepository<User> {
    User findByName(@Param("name") String name);

    @Query("MATCH (u:User) WHERE u.name =~ ('(?i).*'+{name}+'.*') RETURN u")
    Collection<User> findByNameContaining(@Param("name") String name);

}
```

现在可以使用如代码清单 8-16 所示的测试程序来为 Neo4j 数据库生成一些测试数据，以方便后面的测试。测试程序首先清空数据库的记录（如果存在的话），然后创建一个单位

命名为“开发部”，创建两个角色分别是 admin 和 manage，创建一个用户，用户名为 user，将用户“安排”到单位“开发部”中，并给它“分配”两个角色为 admin 和 manage。

代码清单 8-16 Neo4j 测试程序

```
@Autowired
UnitRepository unitRepository;
@Autowired
RoleRepository roleRepository;
@Autowired
UserRepository userRepository;

@Test
public void initData() {
    userRepository.deleteAll();
    roleRepository.deleteAll();
    unitRepository.deleteAll();

    Unit unit = new Unit();
    unit.setName("开发部");
    unit.setCreate(new Date());

    unitRepository.save(unit);
    Assert.notNull(unit.getId());

    Role role = new Role();
    role.setName("admin");
    role.setCreate(new Date());

    roleRepository.save(role);
    Assert.notNull(role.getId());

    Role role1 = new Role();
    role1.setName("manage");
    role1.setCreate(new Date());

    roleRepository.save(role1);
    Assert.notNull(role1.getId());

    User user = new User();
    user.setName("user");
    user.setSex(1);
    user.setEmail("user@email.com");
    user.setCreate(new Date());

    user.beBelong(unit,"安排");
```

```
        user.addOwner(role, "分配");
        user.addOwner(role1, "分配");

        userRepository.save(user);
        Assert.notNull(user.getId());
    }
```

运行测试程序，可以生成一些测试数据。数据生成后，启动 data 模块，然后在浏览器中输入网址：http://localhost:9000/users。

现在可以看到如图 8-3 所示的资源数据，从图中可以看出，可以使用下列链接来访问这些资源数据。

```
http://localhost:9000/users/132
http://localhost:9000/users/search/findByName?name=user
......
```

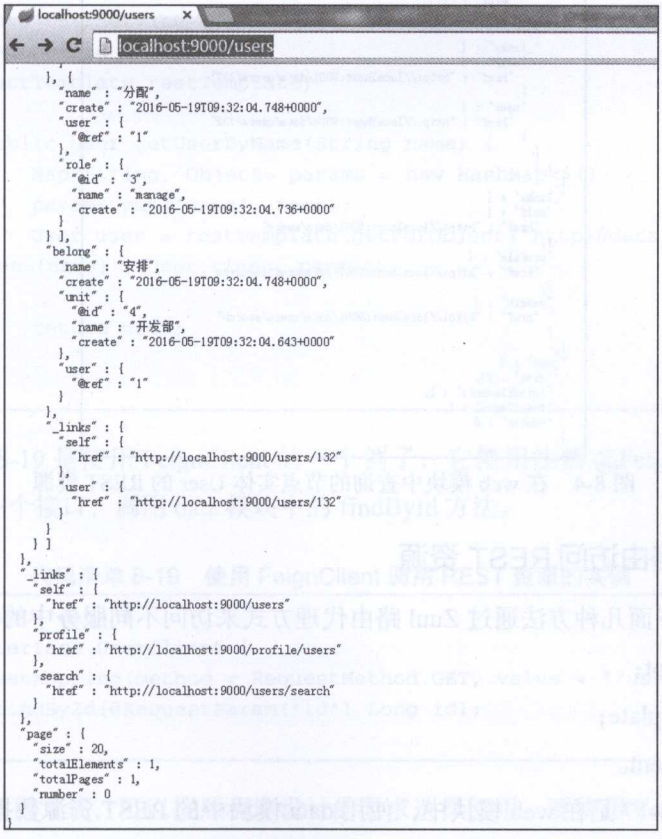


图 8-3　节点实体 User 的 REST 资源

如果这时启动 web 模块，然后使用链接 http://localhost:9001/data/users，同样可以看到类似图 8-3 所示的样子，只不过这时是在 web 模块打开的链接，而不是 data 模块，如图 8-4 所示。也就是说，可以在 web 模块中，像使用本地数据一样使用 data 模块的数据。

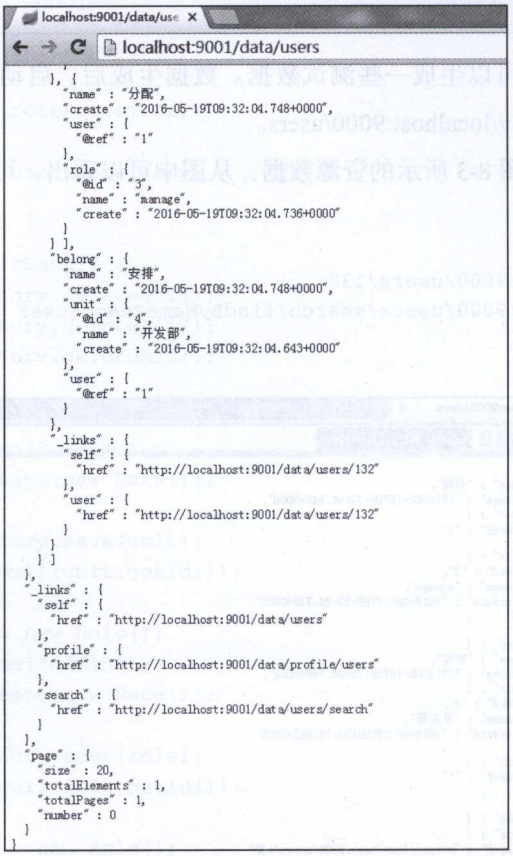


图 8-4 在 web 模块中查询的节点实体 User 的 REST 资源

8.3.3 通过路由访问 REST 资源

可以使用下面几种方法通过 Zuul 路由代理方式来访问不同服务中的 REST 资源。

- JavaScript；
- RestTemplate；
- FeignClient。

代码清单 8-17 是在 web 模块中，使用 data 模块中的 REST 资源数据的一个 jQuery 例子，即将用户名称作为参数，调用 data 模块的 findByNameContaining 使用模糊查询

的方法，返回符合条件的用户列表。

代码清单 8-17　使用 jQuery 调用 REST 资源的例子

```
var pageaction = function(){
    $.get('/data/users/search/findByNameContaining?name='+$("#name").val(),
        function(data){
        var currentData = data["_embedded"].users;
        fillData(currentData);
    });
}
```

代码清单 8-18 是使用 RestTemplate 的一个例子，即同样使用用户名称作为参数，调用 data 模块的 findByName 方法，返回一个用户对象。

代码清单 8-18　使用 RestTemplate 调用 REST 资源的实例

```
@Service
public class UserService {
        @Autowired
        RestTemplate restTemplate;

        public User getUserByName(String name) {
            Map<String, Object> params = new HashMap<>();
            params.put("name", name);
            User user = restTemplate.getForObject("http://data/users/search/
find-ByName?name={name}", User.class, params);

            return user;
        }
}
```

代码清单 8-19 是使用 FeignClient 的一个例子，它使用注解 @FeignClient ("data") 的方式，创建一个接口，调用 data 模块中的 findById 方法。

代码清单 8-19　使用 FeignClient 调用 REST 资源的实例

```
@FeignClient("data")
public interface UserClient {
    @RequestMapping(method = RequestMethod.GET, value = "/users/{id}")
    User findById(@RequestParam("id") Long id);
}
```

上面各种调用方式，可以按照程序设计的要求选择使用。要使用 FeignClient，还必须在工程的 Maven 依赖中增加“ spring-cloud-starter-feign”依赖，并在工程的主程序中

增加一个注解：@EnableFeignClients，以启用 FeignClient 的功能。

8.3.4 使用断路器功能

断路器是微服务中的一个故障容错的线路保护开关。如同电路中的负载保护开关一样，断路器将在所调用的服务过载或出现故障时，自动阻断对服务的访问和调用，转而调用备用方法。

当一个系统服务突然出现故障或超载时，后面访问将会陷入无限地延迟和等待的状态之中，由于请求得不到及时响应，访问者可能会因此而不断发送请求，这将造成严重的恶性循环，最终导致整个系统陷入瘫痪状态，甚至会完全崩溃。使用断路器可以避免这种情况发生，起到防患于未然的作用。

代码清单 8-20 是一个使用断路器和故障容错功能的实例。注解 @HystrixCommand 为 getUserByName 方法配备一个备用方法：getUserFallback。由 getUserFallback 的实现代码可知，该方法只构造了一些虚假数据，以便及时返回请求结果。

代码清单 8-20 使用断路器的实例

```
@Service
public class UserService {
    @Autowired @LoadBalanced
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "getUserFallback")
    public User getUserByName(String name) {
        Map<String, Object> params = new HashMap<>();
        params.put("name", name);
        User user = restTemplate.getForObject("http://data/users/search/findBy
Name?name={name}", User.class, params);

        return user;
    }

    private User getUserFallback(String name) {
        User user = new User();
        user.setName(name + " not find");
        user.setEmail("user email");

        Belong belong = new Belong();
        Unit unit = new Unit();
        unit.setName("unit name");
        belong.setUnit(unit);
```

```
        user.setBelong(belong);

        Owner owner = new Owner();
        Role role = new Role();
        role.setName("role name");
        owner.setRole(role);

        List<Owner> owners = new ArrayList<>();
        owners.add(owner);
        user.setOwners(owners);

        return user;
    }
}
```

8.3.5　路由器和断路器测试

现在可以测试路由器和断路器的功能，首先启动实例工程的 discovery 模块，然后启动实例工程的 data 模块和 web 模块，在浏览器中输入网址 http://localhost:9001/。

返回一个用户列表的界面，这是在 web 服务中请求了 data 服务，并从中取得用户列表数据的结果。在界面中，单击查看，将向 data 服务请求这个用户的详细资料，如果正常，就打开用户的详细信息界面。图 8-5 是一个正常状态的用户信息查看界面。

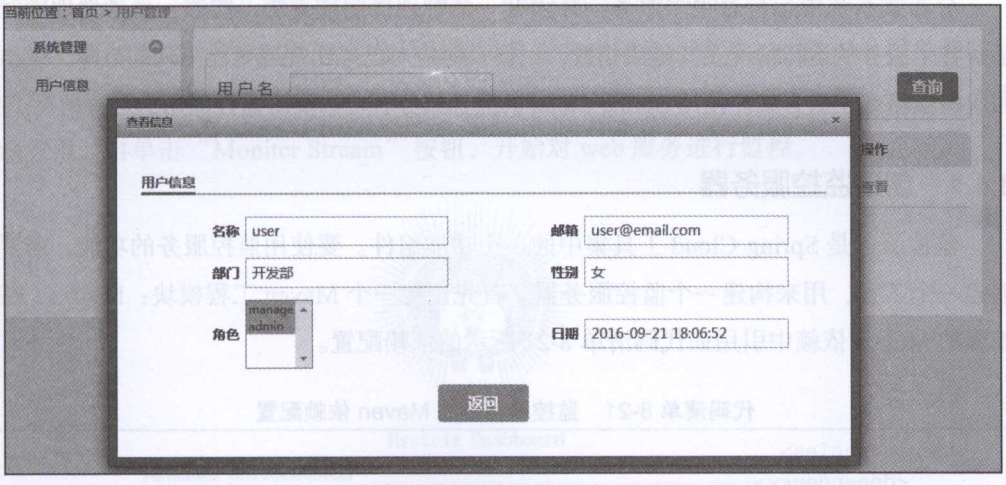


图 8-5　服务正常时查看用户信息的界面

如果这时制造人为的故障，直接关闭 data 服务。现在再在界面中单击查看，它不会陷入无限的等待之中，而是立刻就返回了，只是界面上的用户信息是我们上面编写的

一些虚假数据，如图 8-6 所示。因为 data 服务出现了故障，所以对 data 服务的请求就触发了断路器的熔断机制。

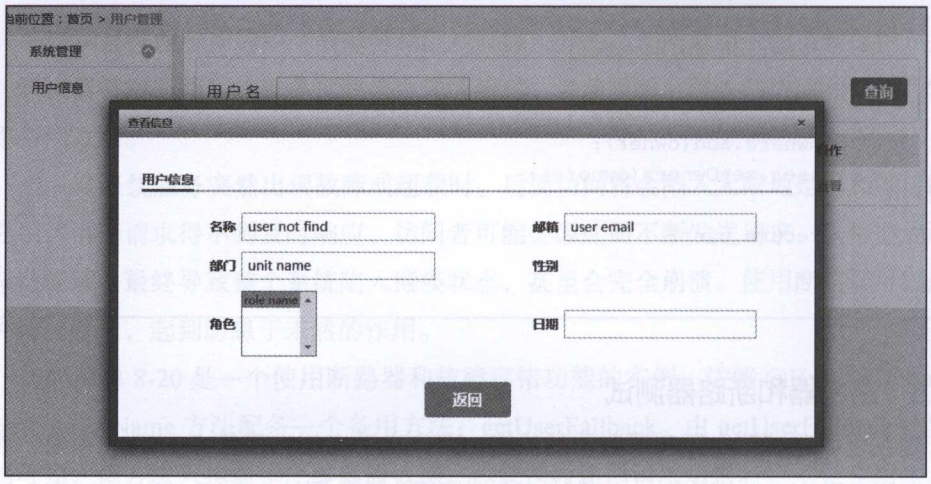


图 8-6 服务故障时查看用户信息的界面

8.4 使用监控服务

分布式系统中运行着很多服务，必须有一个管理机制或方法，能够一目了然地随时了解各个服务的运行情况及其健康指数，以便及时应对已经出现或可能出现的故障，做出相应的策略或改造方案。使用 Spring Cloud 的监控服务，可以实时监控应用的运行情况。

8.4.1 创建监控服务器

监控服务是 Spring Cloud 工具集中的一个功能组件。要使用监控服务的功能，需要创建一个工程，用来构建一个监控服务器。首先创建一个 Maven 工程模块：hystrix，在工程的 Maven 依赖中引用如代码清单 8-21 所示的依赖配置。

代码清单 8-21 监控服务器的 Maven 依赖配置

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
</dependencies>
```

然后创建一个主程序，如代码清单 8-22 所示。程序中使用注解 @EnableHystrix-Dashboard 启用监控功能，使用注解 @Controller 将该程序标注为一个控制器，并设定对根目录的访问，重定向到“/hystrix”位置上。

代码清单 8-22　监控服务主程序

```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class HystrixApplication{
    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, args);
    }
}
```

8.4.2　监控服务器测试

现在可以启动服务测试监控的效果。在实例工程中启动 discovery、hystrix、data、web 四个服务，然后在浏览器中输入网址：http://localhost:7979/。

打开如图 8-7 所示的界面，这是监控服务器的控制台。然后，监控 web 服务的运行情况，在第一行输入框中输入 URL http://localhost:9001/hystrix.stream，其他输入框使用缺省值，再单击“Moniter Stream”按钮，开始对 web 服务进行监控。

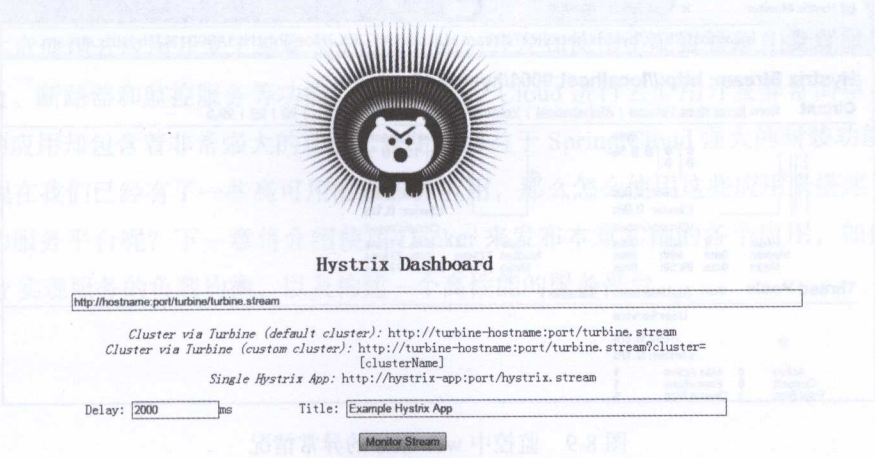


图 8-7　监控服务器首页

这时再按上一节的例子，打开另一个浏览器窗口使用 web 服务，在上面做一些操作，即打开用户列表后，单击查看，打开用户详细信息，等等。返回监控窗口，就可以看到如图 8-8 所示的监控界面。其中 Circuit 中显示了我们请求的服务 data 和请求的方法 getUserByName 及其各种性能指标，Thread Pools 也显示了我们请求的服务：UserService 的情况。

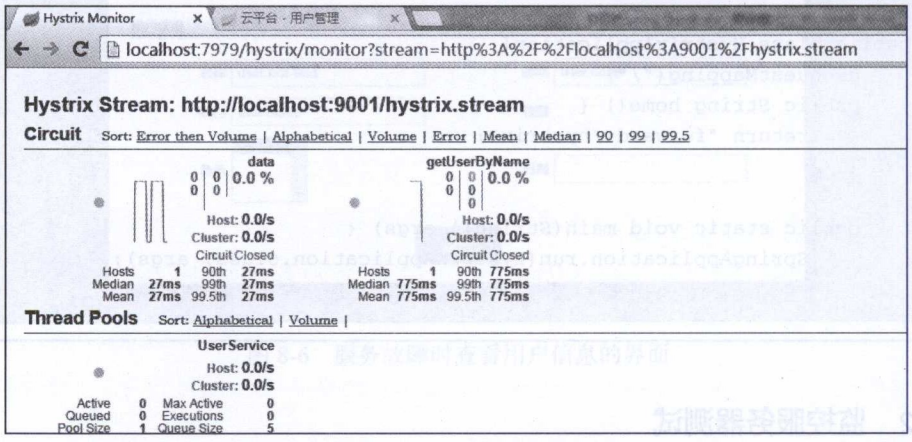


图 8-8 监控中 web 服务的正常情况

再仿照上一节的例子，关闭 data 服务，制造人为的故障，在 web 服务的窗口中单击查看，打开查看用户详细信息的界面，然后返回监控窗口中，就可以看到如图 8-9 所示的情况，这里看到 getUserByName 出现了 100% 的故障。

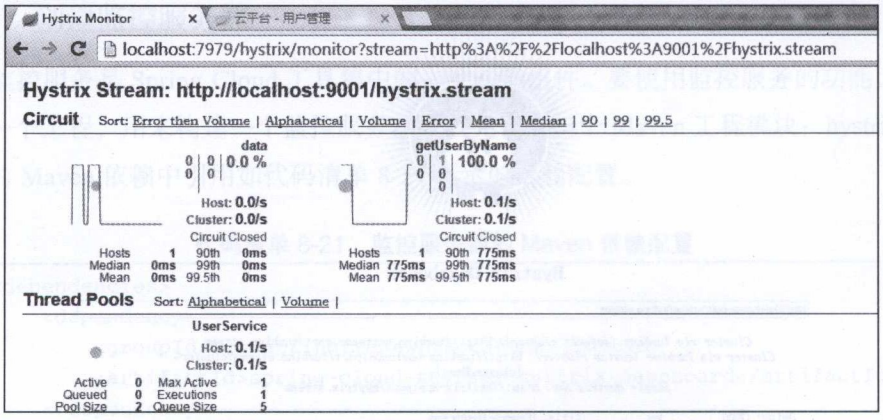


图 8-9 监控中 web 服务的异常情况

8.5　运行与发布

本章实例工程的完整代码可以在 IDEA 中从 GitHub 直接检出：https://github.com/chenfromsz/spring-boot-cloud.git。

检出工程后，在 data 模块的配置文件和测试的 com.test.data.test.Neo4jConfig 配置类中配置好连接 Neo4j 的 URL、用户名和密码。在 IDEA 的 Edit Configuration 中增加一个 Maven 打包配置项目，工作目录选择工程根目录，在命令行中输入：clean package，并将配置保存为 mvn。运行 mvn 配置项目，可以对整个工程进行打包，并自动调用 data 模块的测试程序，生成测试数据。要运行各个模块，可以在命令行窗口使用如下指令：

```
java -jar [模块名称 - 版本].jar
```

也可以在 IDEA 的 Edit Configuration 中为各个模块配置如下 Maven 指令，或者使用命令行窗口切换到各个模块工程所在的目录中，运行下列 Maven 指令，启动各个模块。

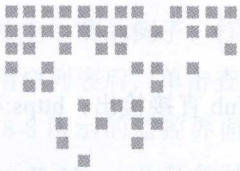
```
mvn spring-boot:run
```

在一台机器上运行以上所有模块工程，将会消耗一定的内存，更改各个模块的配置，将各个模块发布在不同的机器上运行，可以取得较好的运行效果。

8.6　小结

本章使用云应用开发工具集 Spring Cloud，开发和使用了配置管理、发现服务、动态路由、断路器和监控服务等功能。使用 Spring Cloud 进行云应用开发非常简单，但是开发的应用却包含着非常强大的功能，这主要得益于 Spring Cloud 强大的封装功能。

现在我们已经有了一些高可用的微服务应用，那么怎么使用这些应用来搭建一个高性能的服务平台呢？下一章将介绍使用 Docker 来发布本章实例的各个应用，如何使用 Docker 实现服务的负载均衡，以及构建一个高性能的服务平台。



Chapter 9 第 9 章

构建高性能的服务平台

使用 Spring Cloud 开发的微服务，其独立而又相对隔离的特性，与 Docker 的理念有异曲同工之妙，所以使用 Docker 来发布微服务，能够发挥其更大的优势，并且可以非常轻易地构建一个高性能和高可用的服务平台。

Docker 是一个开源的应用容器引擎，可以为应用系统创建一个可移植的独立自主的容器。容器运行在 Linux 操作系统上，相当于一个虚拟机。但 Docker 相比于虚拟机，具有更加明显的优点，它不仅启动很快，而且占用资源极少，所以非常适合用来发布微服务。Docker 将是未来用来发布服务的重要工具，现在许多云服务提供商已经提供了对 Docker 的全力支持。

本书的实例工程都可以非常容易地发布在 Docker 上。要将应用发布在 Docker 上，首先需要在 Docker 中创建应用的镜像，然后就可以使用 Docker 命令来运行应用。为了演示使用 Docker 来发布服务，构建一个高性能的服务平台，将使用第 8 章的实例工程来进行发布和测试。

9.1 使用 Docker

使用 Docker 可以很方便地创建和管理镜像，以及管理已经生成的和正在运行的容器。

那么什么是镜像和容器呢？

镜像是一种文件存储方式，可以把许多文件做成一个镜像文件。例如可以把一个操作

系统做一个 GHOST 镜像，用来重装操作系统，把一个光盘文件做成一个 ISO 镜像，等等。

容器是镜像运行的一个实例。运行一个镜像，就会生成一个容器。容器生成之后，就可以在容器中管理应用系统了。

9.1.1　Docker 安装

在 Linux 上安装 Docker，要求是 64 位系统，并且内核版本需要 3.10 以上，如果使用 CentOS，则使用 CentOS 7.0 可符合要求。可以使用下列指令查看 Linux 的内核版本：

```
# uname -r
```

如果安装了 CentOS 7.0，上述指令将返回类似如下的版本信息：

```
3.10.0-123.el7.x86_64
```

上述版本符合 Docker 的安装要求，可以使用下列指令安装。

首先编写如下内容到 docker.repo 中，以方便 yum 能找到 Docker 引擎：

```
#tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF

# yum update
# yum install docker-engine
```

安装完成后，可以使用下列指令启动 Docker：

```
# service docker start
```

使用下列指令可以查看 Docker 的版本信息：

```
# docker -v
```

如果启动成功，上述指令可以看到版本信息。下列是按照上面安装方法安装后查看的版本信息：

```
Docker version 1.8.2, build bb472f0/1.8.2
```

关于 Docker 的更多安装信息，可以参考其官方网站的说明：https://docs.docker.com/engine/installation/。

9.1.2 Docker 常用指令

Docker 启动之后，就可以使用 Docker 来创建和管理镜像了。下列指令可以测试运行一个已经存在的镜像，它将会生成一个容器并且启动它，然后执行 java –version 指令，最后停止运行的容器。

```
# docker run --name java8 -it java:8 java -version
```

运行上面指令将启动一个包含 Java:8 镜像的容器，如果本地没有这个 Java:8 镜像，将会从远程的镜像服务器中下载一个符合版本号的镜像，执行 Java 的指令打印其版本信息，然后退出容器。但是容器还存在，只是处于停止状态。上面指令将运行 Java 容器并打印出类似如下所示的版本信息：

```
openjdk version "1.8.0_66-internal"
OpenJDK Runtime Environment (build 1.8.0_66-internal-b17)
OpenJDK 64-Bit Server VM (build 25.66-b17, mixed mode)
```

现在，在系统中，存在一个命名为 Java8 的镜像和一个处于停止状态的容器。可以使用 Docker 的一些指令来管理镜像和容器。例如，直接运行这个容器打印出 Java 的版本信息。这个容器没有什么用途，可以将它删除，但这个 Java8 镜像在发布服务时还会用得到。

为了使用 Docker，需要学习 Docker 的一些指令。表 9-1 列出了一些主要指令的功能，将在下节结合实际操作来演示如何使用这些指令。

表 9-1 Docker 主要指令列表

功能	指令	备注
从镜像服务器中查找镜像	docker search < 镜像名 . 版本 >	
拉取镜像	docker pull < 镜像名：tag>	相当于下载
创建镜像	docker build -t < 镜像名 ><Dockerfile 路径 >	需要编写 Dockerfile 生成脚本
查看所有镜像	docker images	
删除镜像	docker rmi < 镜像名 >	
运行一个新容器	docker run –name 容器名 -d -p 内部端口：外部端口 镜像名 <. 版本 >	-a stdin 指定标准输入输出内容类型 -d 后台运行容器并返回 ID； -i 以交互模式运行容器； -t 为容器重新分配一个伪输入终端，通常与 -i 同时使用； --name 为容器指定一个名称； --dns 指定容器使用的 DNS 服务器，默认和宿主一致；

（续）

功能	指令	备注
运行一个新容器	docker run –name 容器名 -d -p 内部端口：外部端口 镜像名 <. 版本 >	--dns-search 指定容器 DNS 搜索域名，默认和宿主一致； -h 指定容器的 hostname； -e username 设置环境变量； --env-file=[] 从指定文件读入环境变量； --cpuset="0-2" or --cpuset="0, 1, 2"
一个容器连接到另一个容器	docker run -i -t --name sonar -d -link mysql:db sonar-server sonar	
查看容器日志	docker logs -f < 容器名或 ID>	
查看正在运行的容器	docker ps	-a 查看所有容器，包括已经停止运行的
删除所有容器	docker rm $ (docker ps -a -q)	
删除单个容器	docker rm < 容器名或 ID>	
停止一个容器	docker stop < 容器名或 ID>	
杀死一个容器	docker kill < 容器名或 ID>	
启动一个容器	docker start < 容器名或 ID>	

9.1.3　使用 Docker 发布服务

为了方便演示，先执行如下指令停掉 CentOS7.0 的防火墙：

```
# systemctl stop firewalld.service #停止 firewall
# systemctl disable firewalld.service #禁止 firewall 开机启动
```

使用第 8 章的实例工程 spring-boot-cloud，可以将各个模块的应用通过 Docker 发布。

首先，将各个模块的配置文件中有 localhsot 的改为安装了 Docker 服务的 Linux 服务器的 IP 地址，因为在测试时发现使用 localhost 连接不了发现服务器。按照 Neo4j 数据库服务器安装的情况，修改模块 data 中连接数据库的参数，按照 RabbitMQ 服务器安装的情况，修改模块 config、模块 web、模块 data 中连接 RabbitMQ 服务器的参数。

然后将整个工程重新打包，这可以在操作系统中打开一个命令行窗口，将路径切换到工程根目录中，输入下列的 Maven 指令执行打包：

```
mvn clean package
```

打包完成后，可将 jar 文件和各个模块中创建镜像的脚本 Dockerfile 上传到安装有 Docker 服务的 Linux 服务器上（实例工程的各个模块已经准备好了创建镜像的脚本，分

别存放在各个模块的 docker 目录下面)。可以在 Linux 服务器中为各个模块创建一个目录，分别用来存放各个模块的 jar 和 Dockerfile 文件。例如，使用如图 9-1 所示的方式创建目录结构。

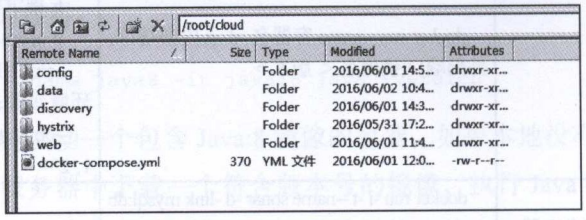


图 9-1 发布服务的目录结构

Dockerfile 是创建镜像的一个脚本文件，它的内容如代码清单 9-1 所示，这是 config 模块创建镜像的脚本，脚本首先导入 java:8 的镜像，最后使用 Java 来运行 jar。其他各个模块的 Dockerfile 跟这个差不多，只是其中的 jar 文件和 EXPOSE 设置不同而已。

代码清单 9-1 创建镜像脚本

```
FROM java:8
VOLUME /tmp
ADD config-1.0-SNAPSHOT.jar app.jar
RUN bash -c 'touch /app.jar'
EXPOSE 8888
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

现在以 config 模块为例，说明如何创建镜像。首先切换到上面上传的 config 模块文件的所在目录，然后执行下列指令，使用当前目录的 Dockerfile 脚本创建一个名称为 config 的镜像，镜像已包含 Java:8 的支持，指令中最后的“.”表示使用当前目录的脚本文件。

```
#docker build -t config .
```

执行上面指令的输出结果如下：

```
Sending build context to Docker daemon 38.57 MB
Step 0 : FROM java:8
 ---> de4a13c84f53
Step 1 : VOLUME /tmp
 ---> Using cache
 ---> 78fdb4381981
Step 2 : ADD config-1.0-SNAPSHOT.jar app.jar
```

```
 ---> 1caac40c3c4b
Removing intermediate container b2ddba948d17
Step 3 : RUN bash -c 'touch /app.jar'
 ---> Running in f032d980aa40
 ---> 2d7145d3875c
Removing intermediate container f032d980aa40
Step 4 : EXPOSE 8888
 ---> Running in e3c025ecb44f
 ---> 10f0835f36b4
Removing intermediate container e3c025ecb44f
Step 5 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar /app.jar
 ---> Running in cb3982402715
 ---> 46dc046aa4e3
Removing intermediate container cb3982402715
Successfully built 46dc046aa4e3
```

从输出结果中可以看到命令执行的步骤，最后完成的镜像 ID 为 46dc046aa4e3。使用下列指令可以查看已经创建的和存在的镜像：

```
#docker images
```

从执行结果可以看出刚才创建的镜像的名称、版本、ID、创建时间和镜像大小等信息，由于没有指定版本，所以默认生成为 latest，如下所示：

```
REPOSITORY    TAG       MAGE ID      CREATED  VIRTUAL SIZE
config        latest     46dc046aa4e3     About a minute ago    719 MB
```

现在可以使用下列指令来运行这个镜像，这个指令设定服务在后台运行，并将容器的内部端口 8888 映射到外部端口 8888，容器的名字也设定为 config。

```
#docker run --name config -d -p 8888:8888 config
```

只有在第一次运行服务时，才需要使用上面的指令，因为它同时会生成一个容器，所以以后需要运行服务时，只要直接启动容器即可。

使用下列指令可以查看正在运行的容器：

```
#docker ps
```

要查看正在运行的容器中服务的运行情况，可以使用下列指令来查看正在运行的服务的控制台输出日志。例如，下列指令可以查看 config 容器的输出日志：

```
#docker logs -f config
```

由于还没有运行发现服务器 discovery，从日志中可以看出输出了一些错误信息。

现在先使用如下指令停止这个容器：

```
#docker stop config
```

要启动一个已经存在的容器，例如启动 config 容器，可以使用如下指令：

```
#docker start config
```

使用上面创建镜像的方法，将实例工程中的其他几个模块，按照模块的名称来命名镜像，各自创建一个镜像。实例工程中每个模块都已经提供了镜像生成脚本 Dockerfile。在下一节将介绍使用更加简便的工具来管理这些镜像及其相关的容器和服务。

所有镜像创建完成之后，使用查看镜像指令可以列出已经创建的镜像，类似于如下所示的镜像列表：

```
REPOSITORY    TAG     IMAGE ID       CREATED          VIRTUAL SIZE
web           latest    da3c3b3da46b       28 seconds ago       721.9 MB
hystrix         latest      2cce1f533f11       About a minute ago    685.4 MB
discovery       latest      cbfa35ccadb9       5 minutes ago         718 MB
data            latest      9ddea4fd1adc       6 minutes ago         825.9 MB
config          latest      46dc046aa4e3       15 hours ago          719 MB
```

9.2 创建和管理一个高性能的服务体系

使用 Docker 发布服务之后，可以使用其他一些服务管理工具来构建高性能和高可用的服务平台。上面创建的镜像可以很方便地使用 docker-compose 来管理。

docker-compose 工具是一个 Docker 容器管理工具集，可以很方便地用来创建和重建容器、执行启动和停止容器等管理操作，以及查看整个服务体系的运行情况和输出日志等。使用 docker-compose 工具，只要一条指令就能启动整个分布式服务体系。

9.2.1 安装 docker-compose

首先，安装 docker-compose 工具，使用下列指令下载已经编译的 docker-compose：

```
#curl -L https://github.com/docker/compose/releases/download/1.7.1/docker-compose
-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

执行指令将下载 docker-compose 可执行文件，并保存在“/user/local/bin”中。这样，可以使用下列指令，赋予它为任何用户可执行：

```
# chmod +x /usr/local/bin/docker-compose
```

这就完成了安装，现在可以使用 docker-compose 指令查看它的版本号：

```
# docker-compose --version
```

如果安装成功，将打印出类似如下的信息：

```
docker-compose 1.7.1
```

9.2.2　docker-compose 常用指令

使用下列指令可以查看 docker-compose 的帮助说明：

```
#docker-compose -h
```

表 9-2 列出了 docker-compose 主要指令的功能。

表 9-2　docker-compose 主要指令列表

功能	指令	备注
创建或重建、启动容器	up	-d 在后台运行
删除停止运行的所有容器	rm	
启动所有容器	start	
停止所有容器	stop	
列出运行的容器列表	ps	
查看正在运行的容器的输出日志	logs	
设置同一个服务运行的容器个数	scale	[service]=[number]
在一个服务上执行一个命令	run	

9.2.3　使用 docker-compose 管理服务

为了使用 docker-compose 将上一节创建的所有镜像，构建成一个服务体系，需要编写一个 docker-compose 模板，如代码清单 9-2 所示。其中 image 用来指定镜像的名称，ports 设定容器的内部和外部端口，links 用来指定一个服务需要依赖的其他服务列表。注意，服务 data 没有指定外部端口，它只供内部调用。

这个 docker-compose 模板将一些高可用的微服务，通过 links 连接成为一个整体，这样就可以通过 docker-compose 统一管理，并且可以在运行的过程中，动态设定服务的负载均衡实例，从而建立一个高性能的服务平台。

代码清单 9-2　docker-compose.yml 文件内容

```
hystrix:
    image: hystrix
    ports:
```

```
    - "7979:7979"
    links:
    - discovery
discovery:
    image: discovery
    ports:
    - "8761:8761"
config:
    image: config
    ports:
    - "8888:8888"
    links:
   - discovery
data:
    image: data
    links:
    - discovery
    - config
web:
    image: web
    ports:
    - "9001:9001"
    links:
    - discovery
    - config
```

将这个模板保存为 docker-compose.yml 文件，并上传到 Linux 服务器上，然后在文件所在位置，执行下列指令，创建如模板所设置的容器，并启动其相应的服务。

```
#docker-compose up
```

在控制台中，可以看到输出日志。从日志中可以看出，docker-compose 按照模板一个一个地创建镜像的容器，然后开始启动服务。

上面指令只有在未创建容器，即第一次运行时，才需要这样执行。当从控制台中退出时，会同时停止已经启动的服务，除非上面指令加上“ -d”参数。生成容器之后，就可以使用如下指令来启动整个服务体系：

```
#docker-compose start
```

在服务运行过程中，如果启动脚本中未设定服务的外部端口，如服务 data，可以使用如下命令指定服务的个数，这个指令将会在线增加一个容器，并且将其自动纳入负载均衡的管理中。

```
#docker-compose scale data=2
```

负载均衡是 Spring Cloud 工具集中一个组件 Ribbon 的功能，这是一个内置了可插拔、可定制的负载均衡组件，它主要具备如下负载均衡规则：

- 简单轮询规则；
- 加权响应时间规则；
- 区域感知轮询规则；
- 随机规则。

Spring Cloud 默认启用 Ribbon 的负载均衡功能，也可以在工程的配置中增加如代码清单 9-3 所示的配置，明确指定启用 Ribbon 的功能。

代码清单 9-3　负载均衡配置

```
ribbon:
    eureka:
        enabled: true
```

现在按照第 8 章的测试方法来测试各个服务的运行情况。注意，为了完成测试，Linux 服务器至少需要 4GB 以上的内存。图 9-2 是发现服务器的控制台，它展示了发现服务器中服务的注册情况。其中 IP 地址 192.168.1.221 是我们使用的 Linux 服务器的 IP，同时可以看到 data 应用已经注册了两个服务，而监控服务 hystrix 因为没有在发现服务器中注册，所以这里看不到。

DS Replicas

192.168.1.221

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG	n/a (1)	(1)	UP (1) - c85938fe9dac:config:8888
DATA	n/a (2)	(2)	UP (2) - ecfb6b82b1d9:data:9000 , 8ca22bd08aac:data:9000
WEB	n/a (1)	(1)	UP (1) - 4a0db0d5b23f:web:9001

General Info

Name	Value
total-avail-memory	368mb
environment	test
num-of-cpus	4
current-memory-usage	152mb (41%)
server-uptime	00:36
registered-replicas	http://192.168.1.221:8761/eureka/
unavailable-replicas	http://192.168.1.221:8761/eureka/,
available-replicas	

图 9-2　服务注册情况

对于两个 data 服务的负载均衡配置，系统只是采用了默认的简单轮询负载均衡规则，当人为停止其中一个 data 服务时，如果不断地调用 getUserByName 方法，会有一瞬间出现 50% 故障的情况，再过一秒左右，系统自动调整以适应只有一个 data 服务的情况，这是 Spring Cloud 的自动修复功能所起的作用，如图 9-3 所示。

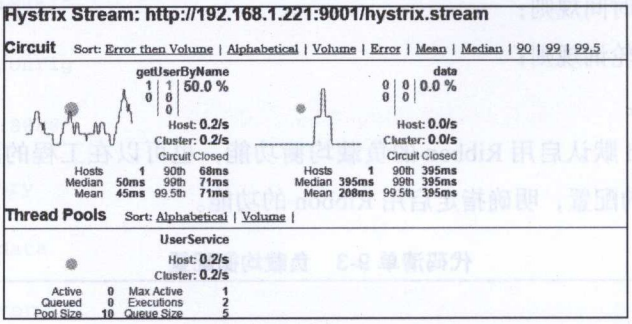


图 9-3 监控服务器中显示出现故障的情况

使用 docker-compose，很轻易地创建了一个高可用和高性能的服务平台，它不但容易管理，而且服务启动速度很快，更加难能可贵的是，它能够动态调整负载均衡配置，实现服务的在线可插拔功能，并且具有自我修复故障的能力。当然，这些功能和性能优势与 Spring Cloud 的支持是分不开的。

这个实例的高可用服务体系结构简图如图 9-4 所示。其中，配置服务器、监控服务器和发现服务器处于整个服务体系的顶层之中，为数据服务和应用实例提供顶端的管理服务功能。而数据服务和应用实例可以使用顶层服务提供的功能，轻易地实现动态路由、断路器和负载均衡等服务。底层的数据服务实例就是由数据服务使用顶层服务提供的负载均衡功能而实现的几个负载均衡的运行实例。

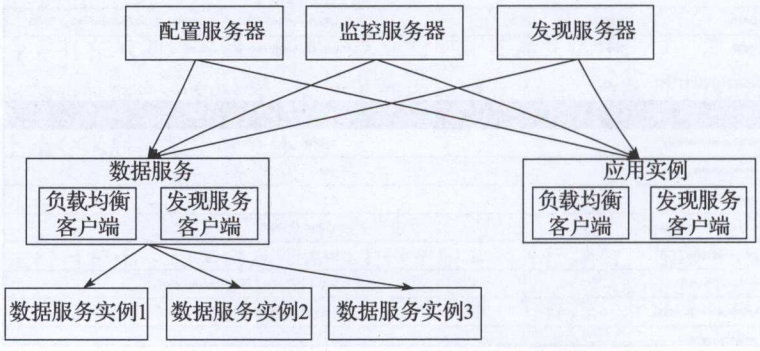


图 9-4 微服务高可用服务体系结构图

9.3　使用 Docker 的其他负载均衡实施方法

Docker 还可以跟其他工具配合使用，配置各种负载均衡服务，搭建高性能的服务平台。例如 Docker 跟 Nginx、Haproxy、Kubernetes 等工具配合使用，都能设计出高性能和高可用的负载均衡服务。不过这些内容已经超过了本书的范围，下面只是简单地介绍，有兴趣的读者可以查找相关的资料进行深入研究。

9.3.1　使用 Nginx 与 Docker 构建负载均衡服务

最简单的莫过于使用 Nginx 来作为负载均衡服务，连接分布于不同机器上的由 Docker 发布的服务。如下代码是一个非常简单的 Nginx 负载均衡配置。

```
server {
    listen 80;
    server_name 192.168.1.10;
    location  / {
        proxy_pass http://blance;
    }
}

upstream blance{
    server 192.168.1.11;
    server 192.168.1.12;
}
```

这个简单的配置是将 Nginx 安装在一台机器上，对外提供服务，它使用负载均衡的机制，将实际使用的服务分布在其他两台机器上。

9.3.2　阿里云的负载均衡设计实例

看看在使用阿里云的负载均衡设计中，使用 Nginx 和 Docker 配置的一个可以动态扩容的负载均衡的样例，这也许能给我们一些参考和启示。

如下代码使用 docker-compose 的模板，配置了两个 Nginx 和两个 Tomcat，并且每个 Tomcat 都运行两个容器。

```
nginx:
    image: 'nginx:latest'
    labels:
        aliyun.routing.port_80: 'http://ngtomcat'
        aliyun.scale: '2'
```

```
    ports:
        - '80'
    links:
        - 'tomcat1:tomcat1'
        - 'tomcat2:tomcat2'
    restart: always
    extra_hosts:
        - "tomcat1.ir:123.56.80.151"
        - "tomcat2.ir:182.92.204.43"

tomcat1:
    environment:
        - LANG=C.UTF-8
        - CATALINA_HOME=/usr/local/tomcat
        - TOMCAT_MAJOR=8
    image: 'tomcat:latest'
    labels:
        aliyun.scale: '2'
        aliyun.routing.port_8080: 'http://tomcat.ir'
    ports:
        - '8080'
    restart: always

tomcat2:
    environment:
        - LANG=C.UTF-8
        - CATALINA_HOME=/usr/local/tomcat
        - TOMCAT_MAJOR=8
    image: 'tomcat:latest'
    labels:
        aliyun.scale: '2'
        aliyun.routing.port_8080: 'http://tomcat.ir'
    ports:
        - '8080'
    restart: always
```

而它的 Nginx 的配置如下代码所示：

```
upstream tomcat.ir {
    server tomcat.ir.1;
    server tomcat.ir.2;
}

server {
listen       80;
server_name tomcat.ir;
```

```
index index.html index.htm index.php;
access_log  /var/log/nginx/access.log;
location / {
proxy_pass  http://tomcat.ir;
}
}
```

（上面资料来源：https://yq.aliyun.com/articles/6816#3 ）

9.4　小结

本章介绍使用 Docker 来发布分布式应用系统，演示了使用一个更加轻量级的容器管理工具 docker-compose 来构建一个高性能和高可用的服务平台。在演示中，我们发现 Docker 的运行速度非常流畅，这是虚拟机系统无法比拟的，使用 Docker 来发布和管理服务将是未来的发展趋势。此外，Docker 还可以跟其他负载均衡工具配合使用，以搭建一个高性能和高可用的服务平台。

在后续的章节中，将分析 Spring Boot 的一些核心功能的源代码，以加深对 Spring Boot 的理解，还能帮助我们为更好地使用 Spring Boot 框架铺平道路。

第三部分 Part 3

核心技术源代码分析

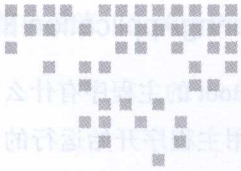
- 第 10 章　Spring Boot 自动配置实现原理
- 第 11 章　Spring Boot 数据访问实现原理
- 第 12 章　微服务核心技术实现原理

这一部分将简要分析 Spring Boot 的一些核心功能的源代码及其实现原理，加深对 Spring Boot 的理解和学会如何更好地使用 Spring Boot：

第 10 章分析 Spring Boot 应用中程序入口的源代码、Spring Boot 自动配置的实现原理，同时利用自动配置的原理，演示如何在主程序中通过更改加载配置的方式，提升应用的性能。

第 11 章简要分析 Spring Boot 访问数据库的源代码和实现原理，并在探索其实现原理的过程中，扩展访问数据库的功能。

第 12 章简要分析微服务中配置管理、发现服务和负载均衡服务的源代码和实现原理，同时使用一个简单的例子，形象地说明了微服务中使用分布式消息的实现原理。



第 10 章 Chapter 10

Spring Boot 自动配置实现原理

通过前面章节的学习，我们掌握了使用 Spring Boot 框架进行实际应用开发的方法。在使用 Spring Boot 的过程中，我们时常会为一些看似简单，但实际上蕴藏了强大功能的实现而惊呼，下面就让我们来揭开它的神秘面纱，做到知其然，进而知其所以然。在认识 Spring Boot 的实现原理之后，我们在使用某些功能时，就能够做到心中有数，从而更好地使用它。

10.1 Spring Boot 主程序的功能

代码清单 10-1 是 Spring Boot 应用的一个典型主程序，它看起来非常简单，使用了一个同样看起来非常简单的注解 @SpringBootApplication，并用一个非常普通的 main 方法运行 SpringApplication 的 run 方法。这个简单的主程序将会加载一个应用所需的所有资源和配置，最后启动一个应用实例。

代码清单 10-1 Spring Boot 应用主程序

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

10.1.1 SpringApplication 的 run 方法

Spring Boot 的主程序有什么神奇的地方呢？可以从 SpringApplication 的 run 方法说起，这是应用主程序开始运行的方法，它的源代码如代码清单 10-2 所示。这个方法让我们能够看清楚 Spring Boot 的一切秘密所在，它首先开启一个 SpringApplicationRunListeners 监听器，然后创建一个应用上下文 ConfigurableApplicationContext，通过这个上下文加载应用所需的类和各种环境配置等，最后启动一个应用实例。

代码清单 10-2 SpringApplication 中 run 的源代码

```
package org.springframework.boot;
......
public class SpringApplication {
    ......
    public ConfigurableApplicationContext run(String... args) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        ConfigurableApplicationContext context = null;
        this.configureHeadlessProperty();
        SpringApplicationRunListeners listeners = this.getRunListeners(args);
        listeners.started();

        try {
            DefaultApplicationArguments ex = newDefaultApplicationArguments(args);
            context = this.createAndRefreshContext(listeners, ex);
            this.afterRefresh(context, (ApplicationArguments)ex);
            listeners.finished(context, (Throwable)null);
            stopWatch.stop();
            if(this.logStartupInfo) {
                (new StartupInfoLogger(this.mainApplicationClass)).logStarted
(this.getApplicationLog(), stopWatch);
            }

            return context;
        } catch (Throwable var6) {
            this.handleRunFailure(context, listeners, var6);
            throw new IllegalStateException(var6);
        }
    }
    ......
}
```

如果你使用过 Spring 框架，就会更加清楚这种加载应用的实现机制。在 Spring 中，

加载一个应用，主要是通过一些复杂的配置来实现的。这样看来，Spring Boot 只不过是把这些本来由程序员做的工作，事先帮我们实现罢了。

10.1.2　创建应用上下文

一个应用能够正常运行起来，需要一些环境变量、各种资源和一些相关配置等，从创建应用上下文 ConfigurableApplicationContext 的源代码中，我们可以看到这种实现机制，如代码清单 10-3 所示。其中，this.load (context, sources.toArray (new Object[sources.size()])) 将调用 BeanDefinitionLoader 来加载应用定义的和需要的类及各种资源。

代码清单 10-3　创建应用上下文——createAndRefreshContext 的源代码

```
package org.springframework.boot;
......
public class SpringApplication {
    ......
    private ConfigurableApplicationContext createAndRefreshContext(SpringApp
licationRunListeners listeners, ApplicationArguments applicationArguments) {
        ConfigurableEnvironment environment = this.getOrCreateEnvironment();
        this.configureEnvironment(environment, applicationArguments.getSource
Args());
        listeners.environmentPrepared(environment);
        if(this.isWebEnvironment(environment) && !this.webEnvironment) {
            environment = this.convertToStandardEnvironment(environment);
        }

        if(this.bannerMode != Mode.OFF) {
            this.printBanner(environment);
        }

        ConfigurableApplicationContext context = this.createApplicationContext();
        context.setEnvironment(environment);
        this.postProcessApplicationContext(context);
        this.applyInitializers(context);
        listeners.contextPrepared(context);
        if(this.logStartupInfo) {
            this.logStartupInfo(context.getParent() == null);
            this.logStartupProfileInfo(context);
        }

        context.getBeanFactory().registerSingleton("springApplicationArguments",
applicationArguments);
        Set sources = this.getSources();
```

```
            Assert.notEmpty(sources, "Sources must not be empty");
            this.load(context, sources.toArray(new Object[sources.size()]));
            listeners.contextLoaded(context);
            this.refresh(context);
            if(this.registerShutdownHook) {
                try {
                    context.registerShutdownHook();
                } catch (AccessControlException var7) {
                    ;
                }
            }

            return context;
        }
        ......
    }
```

10.1.3 自动加载

在 BeanDefinitionLoader 中，有一个 load (Class<?> source) 方法用来加载类定义，如代码清单 10-4 所示。这里的 source 就是代码清单 10-1 中定义的 Application.class。在程序中通过 isComponent 检查是否存在注解，如果有注解，则调用注解相关的类定义。这样注解 @SpringBootApplication 将被调用，它不但会导入一系列自动配置的类，还会加载应用中一些自定义的类。

代码清单 10-4 BeanDefinitionLoader 中 load(Class<?> source) 源代码

```
package org.springframework.boot;
......
class BeanDefinitionLoader {
    ......
    private int load(Class<?> source) {
        if(this.isGroovyPresent() && BeanDefinitionLoader.GroovyBeanDefinition
Source.class.isAssignableFrom(source)) {
            BeanDefinitionLoader.GroovyBeanDefinitionSource loader = (Bean
DefinitionLoader.GroovyBeanDefinitionSource)BeanUtils.instantiateClass(source,
BeanDefinitionLoader.GroovyBeanDefinitionSource.class);
            this.load(loader);
        }

        if(this.isComponent(source)) {
            this.annotatedReader.register(new Class[]{source});
            return 1;
```

```
            } else {
                return 0;
            }
        }
        ......
        private boolean isComponent(Class<?> type) {
            return AnnotationUtils.findAnnotation(type, Component.class) != null?
true:!type.getName().matches(".*\\$_.*closure.*") && !type.isAnonymousClass()
&& type.getConstructors() != null && type.getConstructors().length != 0;
        }
        ......
    }
```

从以上分析可知，一个简单的 Spring Boot 主程序，通过运行一个 run 方法，就将引发一系列复杂的内部调用和加载过程，从而初始化一个应用所需的配置、环境、资源及各种类定义等。特别是导入了一系列自动配置类，实现了强大的自动配置功能，这是 Spring Boot 框架最引人注目的地方。

10.2　Spring Boot 自动配置原理

所有的自动配置都是从注解 @SpringBootApplication 引入的，我们来看看它的源代码，就一切都明白了。如代码清单 10-5 所示，注解 @SpringBootApplication 其实又包含了三个非常重要的注解，即 @Configuration、@EnableAutoConfiguration 和 @ComponentScan，其中注解 @EnableAutoConfiguration 就是启用自动配置的，并将导入一些自动配置的类定义，注解 @ComponentScan 将扫描和加载应用中的一些自定义的类。

代码清单 10-5　SpringBootApplication 源代码

```
package org.springframework.boot.autoconfigure;
......
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {
    ......
}
```

10.2.1 自动配置的即插即用原理

EnableAutoConfiguration 最终会导入一个自动配置的类列表，如代码清单 10-6 所示。列表中的自动配置类很多，这些配置类中大都将被导入，并处于备用状态中，这如同电器中准备了一些插槽一样，即实现了即插即用的原理。这样，当项目中引入了相关的包时，相关的功能将被启用。例如在项目的 Maven 管理中配置了 Redis 的引用，那么 Redis 的功能将被启用，这时启动应用，程序将尝试读取有关 Redis 的配置信息。

代码清单 10-6 自动配置类部分列表

```
    # Auto Configure
    org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAuto
Configuration,\
    org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
    org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
    org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
    org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
    org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
    org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
    org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
    org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
    org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAuto
Configuration,\
    org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslation
AutoConfiguration,\
    org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoCon
figuration,\
    org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositorie
sAutoConfiguration,\
    org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAut
oConfiguration,\
    org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDat
aAutoConfiguration,\
    org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRep
ositoriesAutoConfiguration,\
    org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfigu
ration,\
    org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
    org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoCon
figuration,\
    org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfi
guration,\
```

```
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
......
```

10.2.2　自动配置的约定优先原理

在自动配置中加载一个类的配置时，首先读取项目中的配置，只有项目中没有相关配置才启用配置的默认值，这就是自动配置的约定优先原理。代码清单 10-7 是 Thymeleaf 配置类的源代码，如果在项目的配置文件中没有配置 spring.thymeleaf 的相关参数，就使用 Thymeleaf 的默认配置，默认配置将使用 templates 作为 HTML 文件的存放路径。在前面章节使用 Thymeleaf 的实例中，就是使用了这个默认配置。

代码清单 10-7　Thymeleaf 配置源代码

```
package org.springframework.boot.autoconfigure.thymeleaf;
......
@ConfigurationProperties("spring.thymeleaf")
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING = Charset.forName("UTF-8");
    private static final MimeType DEFAULT_CONTENT_TYPE = MimeType.valueOf("text/
html");
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
    private boolean checkTemplateLocation = true;
    private String prefix = "classpath:/templates/";
    private String suffix = ".html";
    private String mode = "HTML5";
    private Charset encoding;
    private MimeType contentType;
    private boolean cache;
    private Integer templateResolverOrder;
    private String[] viewNames;
    private String[] excludedViewNames;
    private boolean enabled;
    ......
}
```

10.3　提升应用的性能

Spring Boot 的自动配置在给我们提供很大便利的同时，难免会有一些副作用，即增加了应用启动的时间、一些内存和 CPU 的消耗等。如果应用对性能要求很高，就可以根据自动配置的原理，使用一些技巧进行优化。

10.3.1 更改加载配置的方式

如果能清楚一个应用需要哪些配置，就能够更改加载配置的方式，即不使用自动配置，而是改为指定加载一些应用所需的配置。

为了弄清楚一个应用需要加载哪些配置，可以使用 Maven 调试的方式来启动一个应用，然后从控制台的输出日志中，确定哪些是这个应用需要加载的配置类。下面使用第 1 章中简单的实例项目来说明这种操作。

首先，在 IDEA 的 Edit Configuration 中增加一个 Maven 配置，工作路径选择项目根目录，在命令行中输入：spring-boot:run –Ddebug，并把配置保存为 debug，如图 10-1 所示。

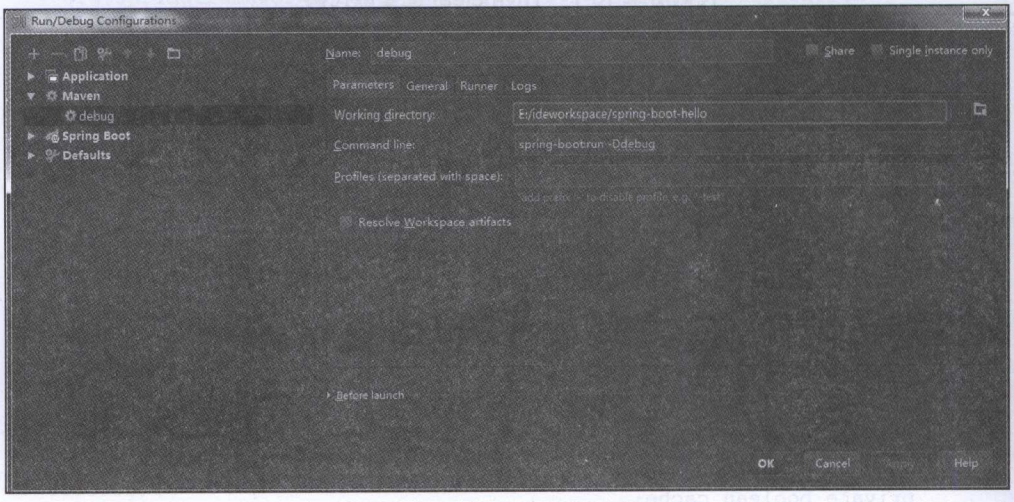


图 10-1 使用 Maven 调试的配置

以 Debug 方式运行 debug 配置，启动应用，然后在控制台中找出 Positive matches 的类，如代码清单 10-8 所示。Positive matches 就是这个应用所需加载的一些配置类。

代码清单 10-8 加载自动配置的 Positive matches 类列表

```
=========================
AUTO-CONFIGURATION REPORT
=========================
Positive matches:
-----------------
   DispatcherServletAutoConfiguration matched
      - @ConditionalOnClass classes found: org.springframework.web.servlet.
```

```
DispatcherServlet(OnClassCondition)
            - found web application StandardServletEnvironment (OnWebApplication
Condition)
    ......
```

通过整理后，得出这个应用需要加载的配置类列表，如代码清单 10-9 所示。

代码清单 10-9　整理后的 Positive matches 类列表

```
Positive matches:
-----------------
    DispatcherServletAutoConfiguration matched
    EmbeddedServletContainerAutoConfiguration matched
    GenericCacheConfiguration matched
    HttpEncodingAutoConfiguration matched
    HttpMessageConvertersAutoConfiguration matched
    JacksonAutoConfiguration matched
    JmxAutoConfiguration matched
    MultipartAutoConfiguration matched
    NoOpCacheConfiguration matched
    RedisCacheConfiguration matched
    ServerPropertiesAutoConfiguration matched
    SimpleCacheConfiguration matched
    ThymeleafAutoConfiguration matched
    WebMvcAutoConfiguration matched
    WebSocketAutoConfiguration matched
```

根据这个配置类加载列表，就可以在主程序中使用注解 @Configuration 来代替注解 @SpringBootApplication，并用注解 @Import 指定需要加载的配置类，经过更改后的应用主程序如代码清单 10-10 所示。

代码清单 10-10　主程序中指定加载的配置类

```
@Configuration
@Import({
        DispatcherServletAutoConfiguration.class,
        EmbeddedServletContainerAutoConfiguration.class,
        ErrorMvcAutoConfiguration.class,
        HttpEncodingAutoConfiguration.class,
        HttpMessageConvertersAutoConfiguration.class,
        JacksonAutoConfiguration.class,
        JmxAutoConfiguration.class,
        MultipartAutoConfiguration.class,
        ServerPropertiesAutoConfiguration.class,
        PropertyPlaceholderAutoConfiguration.class,
```

```
        ThymeleafAutoConfiguration.class,
        WebMvcAutoConfiguration.class,
        WebSocketAutoConfiguration.class
})
@RestController
public class Application {
    @RequestMapping("/")
    String home() {
        return "hello";
    }
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

10.3.2 将 Tomcat 换成 Jetty

另外，为了提高应用的性能，还可以更改默认使用的 Tomcat 插件，换成更加小巧的 Jetty 插件。例如，代码清单 10-11 是在工程的 Maven 配置中排除引用默认的 Tomcat，转而引用 Jetty 的依赖。

代码清单 10-11 使用 Jetty 的 Maven 配置

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jetty</artifactId>
        </dependency>

</dependencies>
```

10.4　性能对照测试

通过上面一些改造之后，可以对照测试一下，看看效果如何。打开 IDEA 的 Edit Configuration 对话框，增加一个 Application 配置，工作目录选择工程根目录，并选择工程主程序，然后在 VM options 中输入如下配置参数：

```
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port="9004"
-Dcom.sun.management.jmxremote.authenticate="false"
-Dcom.sun.management.jmxremote.ssl="false"
```

这样配置的目的，是让我们可以使用 JConsole 来观察应用运行的各项性能指标。配置完成后的效果如图 10-2 所示。

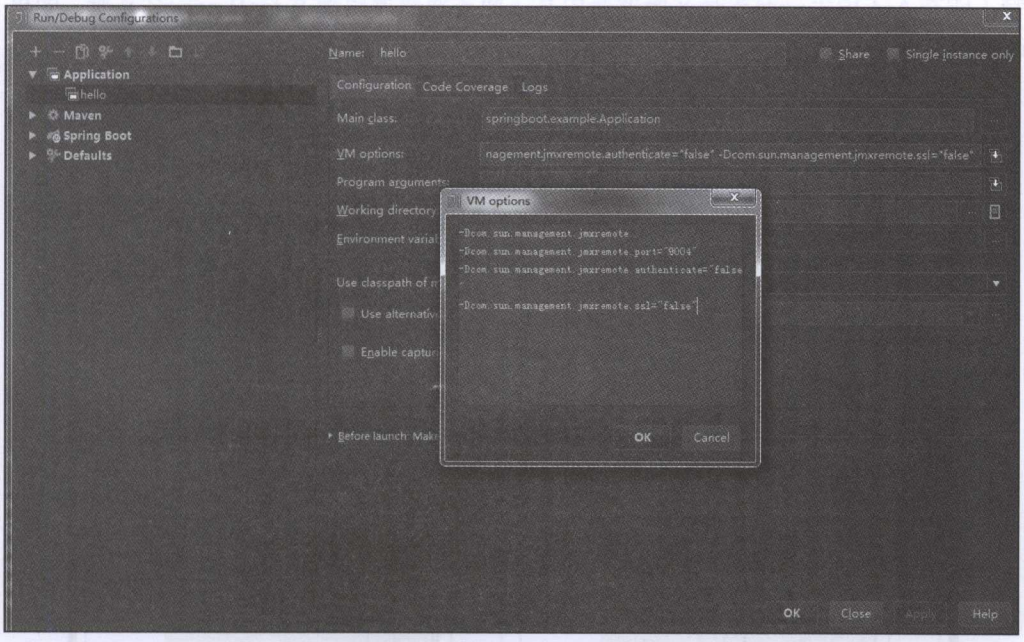


图 10-2　启动应用测试配置

对比改造前后的两种情况，改造后应用的启动时间有所加快。

改造前启动应用的时间如下所示：

```
Started Application in 3.171 seconds (JVM running for 4.941)
```

改造后启动应用的时间如下：

```
Started Application in 2.957 seconds (JVM running for 5.869)
```

应用启动后，使用 JConsole 新建一个连接，可以观察应用运行的各项性能指标。根据上面配置的参数，可以在远程进程中输入 localhost:9004，然后单击“连接”按钮，如图 10-3 所示。

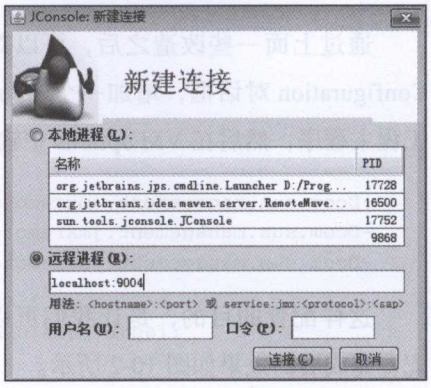


图 10-3 JConsole 新建连接

改造前后的两种运行情况对照如图 10-4 所示。图中各项指标处于 0 的位置是中间停止时的状态，从图中可以看出，改造后内存的使用量明显减少了，CPU 的占用也有所改善，加载的类减少了一点，并不是很明显。从总体上来说，性能是有所改善了。

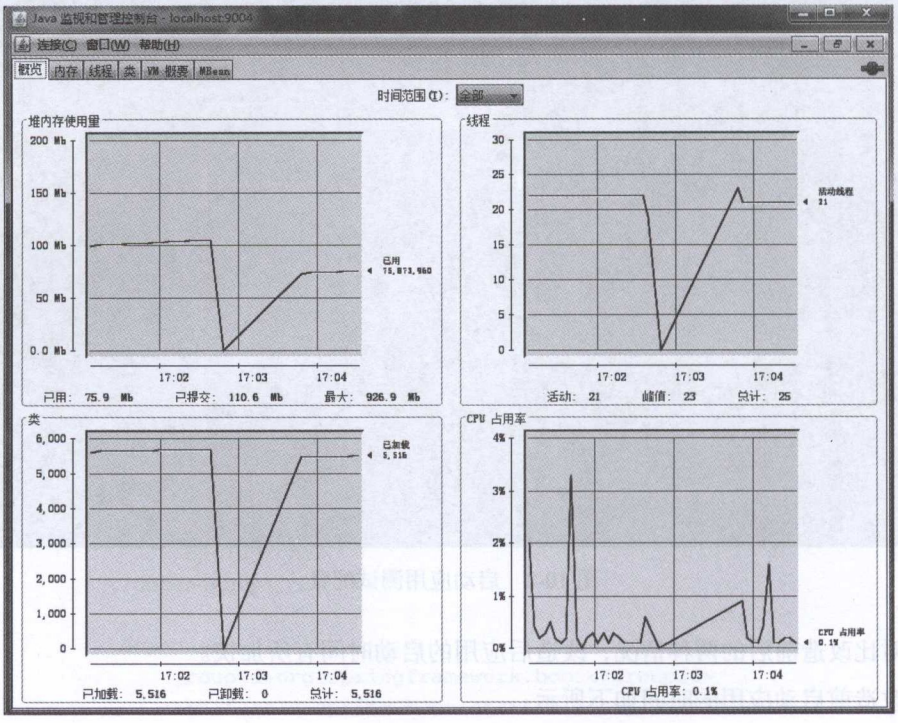


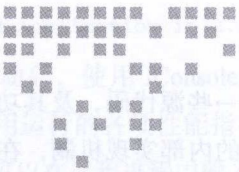
图 10-4 改造前后的两种运行情况对照图

10.5　小结

本章分析了 Spring Boot 应用主程序的内部实现的一些源代码，及其功能强大的自动配置的实现原理，使我们认识了神奇的 Spring Boot 的内部实现机制，在看似简单的调用中，其实包含着复杂的内部实现。这就不难理解，为什么使用 Spring Boot 可以那么简单，这是因为它把一些复杂的实现，都事先帮我们做好了。

基于对 Spring Boot 的深入了解，特别是认识自动配置的实现原理之后，就可以改造一个应用加载配置的方式，从而达到提高性能的目的。虽然这种改造的作用并不是特别明显，但是不管怎样，至少能帮助我们加深对 Spring Boot 的理解。

通过前面章节的应用实例，我们也知道，在 Spring Boot 中使用数据库也是非常简单的，那么 Spring Boot 在使用数据库方面，其内部实现又是怎样一个引人入胜的工程呢？下一章将分析 Spring Boot 在使用数据库方面的一些实现原理，看看它又有什么神奇之处。



Chapter 11 第 11 章

Spring Boot 数据访问实现原理

Spring Boot 的数据库管理功能非常强大，它到底可以支持哪些数据库，访问这些数据库的功能又是如何实现的，这些功能有没有欠缺或者需要扩展的地方，如果需要扩展应该如何改进？如果你对这些问题感兴趣，那么可以开始这一章的学习。

11.1 连接数据源的源代码分析

要使用数据库，首先必须与数据库服务器建立连接。对于关系型数据库，Spring Boot 连接数据源一般都采用 JDBC（Java Data Base Connectivity）的方式来实现。其他类型的数据库却使用各自独立的方式来建立连接。代码清单 11-1 是 JDBC 的一些连接参数的定义，使用关系型数据库时，Spring Boot 的自动配置将尝试从应用的配置文件中读取这些配置项。

代码清单 11-1　数据源配置参数定义源代码

```
package org.springframework.boot.autoconfigure.jdbc;
......
@ConfigurationProperties(
    prefix = "spring.datasource"
)
public class DataSourceProperties implements BeanClassLoaderAware, EnvironmentAware,
InitializingBean {
    public static final String PREFIX = "spring.datasource";
```

```
    private ClassLoader classLoader;
    private Environment environment;
    private String name = "testdb";
    private Class<? extends DataSource> type;
    private String driverClassName;
    private String url;
    private String username;
    private String password;
    private String jndiName;
    private boolean initialize = true;
    private String platform = "all";
    private String schema;
    private String data;
    private boolean continueOnError = false;
    private String separator = ";";
    private Charset sqlScriptEncoding;
    private EmbeddedDatabaseConnection embeddedDatabaseConnection;
    private DataSourceProperties.Xa xa;
    ......
}
```

11.1.1　数据源类型和驱动

JDBC 连接数据源必须指定数据源类型和数据库驱动程序，在程序中用 driverClassName 来指定数据库驱动程序的名称，例如使用 MySQL 的驱动程序是 com.mysql.jdbc.Driver，而数据源的类型默认使用 org.apache.tomcat.jdbc.pool.DataSource，如代码清单 11-2 所示。

代码清单 11-2　JDBC 的 DataSourceBuilder 源代码片段

```
package org.springframework.boot.autoconfigure.jdbc;
    ......
public class DataSourceBuilder {
    private static final String[] DATA_SOURCE_TYPE_NAMES = new String[]{"org.apache.
tomcat.jdbc.pool.DataSource", "com.zaxxer.hikari.HikariDataSource", "org.apache.
commons.dbcp.BasicDataSource", "org.apache.commons.dbcp2.BasicDataSource"};
    private Class<? extends DataSource> type;
    private ClassLoader classLoader;
    private Map<String, String> properties = new HashMap();
        ......
    public Class<? extends DataSource> findType() {
        if(this.type != null) {
            return this.type;
        } else {
```

```
            String[] var1 = DATA_SOURCE_TYPE_NAMES;
            int var2 = var1.length;
            int var3 = 0;

            while(var3 < var2) {
                String name = var1[var3];

                try {
                    return ClassUtils.forName(name, this.classLoader);
                } catch (Exception var6) {
                    ++var3;
                }
            }

            return null;
        }
    }
}
```

数据源的类型可以通过配置更改，这一功能给我们使用其他数据源提供了很大的便利。例如在第 4 章中使用 Druid 数据源时，就是使用如下的配置来指定数据源类型，即代码中的 type 设定为 com.alibaba.druid.pool.DruidDataSource。

```
spring:
    datasource:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: com.mysql.jdbc.Driver
        ......
```

11.1.2 支持的数据库种类

Spring Boot 默认几乎可以支持现有的所有数据库，代码清单 11-3 是 DatabaseDriver 定义一个枚举类的源代码，从这个数据库驱动的定义列表中可以看出，它默认支持的数据库种类。但这并不是说，这个列表中没有列出的数据库就不支持了，只是可以采用其他方式来建立连接而已。例如，Redis、MongoDB、Neo4j 等数据库都使用了各自独特的方式来建立连接。

代码清单 11-3 枚举类 DatabaseDriver 源代码片段

```
package org.springframework.boot.autoconfigure.jdbc;
......
enum DatabaseDriver {
    UNKNOWN((String)null),
    DERBY("org.apache.derby.jdbc.EmbeddedDriver"),
```

```
        H2("org.h2.Driver", "org.h2.jdbcx.JdbcDataSource"),
        HSQLDB("org.hsqldb.jdbc.JDBCDriver", "org.hsqldb.jdbc.pool.JDBCXADataSource"),
        SQLITE("org.sqlite.JDBC"),
        MYSQL("com.mysql.jdbc.Driver", "com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"),
        MARIADB("org.mariadb.jdbc.Driver", "org.mariadb.jdbc.MySQLDataSource"),
        GOOGLE("com.google.appengine.api.rdbms.AppEngineDriver"),
        ORACLE("oracle.jdbc.OracleDriver", "oracle.jdbc.xa.client.OracleXADataSource"),
        POSTGRESQL("org.postgresql.Driver", "org.postgresql.xa.PGXADataSource"),
        JTDS("net.sourceforge.jtds.jdbc.Driver"),
        SQLSERVER("com.microsoft.sqlserver.jdbc.SQLServerDriver", "com.microsoft.sqlserver.jdbc.SQLServerXADataSource"),
        DB2("com.ibm.db2.jcc.DB2Driver", "com.ibm.db2.jcc.DB2XADataSource"),
        AS400("com.ibm.as400.access.AS400JDBCDriver", "com.ibm.as400.access.AS400JDBCXADataSource");

        private final String driverClassName;
        private final String xaDataSourceClassName;
        .....
    }
```

11.1.3　与数据库服务器建立连接

最终，不管使用哪一种数据库，与数据库服务器建立连接大体上都会用到三个参数，即数据库链接地址、用户名和密码。下面我们来看一看，Spring Boot 在使用一个新型的数据库 Neo4j 时是怎么建立连接的。Neo4j 是一个 NoSQL 数据库，它不能使用 JDBC 的方式来建立连接，所以由 spring-data-neo4j 提供连接数据库服务器的方法。使用 Neo4j 数据库，一般使用如代码清单 11-4 所示的方法来连接数据库服务器，即通过继承 Neo4jConfiguration，重载 neo4jServer 方法来实现。

代码清单 11-4　连接 Neo4j 的配置类定义

```
@Configuration
......
public class Neo4jConfig extends Neo4jConfiguration {
    @Override
    public Neo4jServer neo4jServer() {
        return new RemoteServer("http://192.168.1.221:7474","neo4j","12345678");
    }
    ......
}
```

上面的代码最终将调用超类 Neo4jConfiguration 中的 constructSession 方法，使用提

供的链接地址、用户名和密码，与数据库服务器建立连接，如代码清单 11-5 所示。

代码清单 11-5 Neo4j 连接服务器部分源代码

```
package org.springframework.data.neo4j.config;
......
@Configuration
public abstract class Neo4jConfiguration {
    private Session constructSession(Neo4jServer server) {
            return server.url() != null && server.username() != null && server.
password() != null?this.getSessionFactory().openSession(server.url(), server.
username(), server.password()):this.getSessionFactory().openSession(server.url());
    }
    ......
}
```

11.2 数据存取功能实现原理

与数据库服务器建立连接之后，就可以对数据库执行一些存取操作，对数据库实现管理的功能。数据存取的操作大体上都包含两个方面的内容，即实体建模和持久化。不管是关系型数据库，还是 NoSQL 数据库，都遵循这一设计规范。

11.2.1 实体建模源代码分析

实体建模的原理简单地说，即将 Java 的普通对象和关系映射为数据库表及其相关的关系。而这种映射在 Spring Boot 中，主要是通过注解的方式来实现。几种数据库中主要的注解定义如表 11-1 所示。

表 11-1 实体建模主要注解定义

数据库	类型	主要注解
MySQL	关系型	@Entity @Table @Id @GeneratedValue @ManyToOne @ManyToMany @JoinTable @JoinColumn
MongoDB	NoSQL	@Document @Id @Indexed @Language

（续）

数据库	类型	主要注解
Neo4j	NoSQL	@NodeEntity @GraphId @Relationship @Index @Property

这种映射机制是双向的，当向数据库存入数据时，是将 Java 对象映射为数据库对象，而从数据库取出数据时，却将数据库中的数据还原为 Java 对象。

例如对于 Neo4j 来说，在实体建模中的主要注解 @NodeEntity 的定义如代码清单 11-6 所示。在一个类定义中使用这个注解，表示这个类定义就是一个节点实体的建模。

代码清单 11-6　@NodeEntity 源代码

```
package org.neo4j.ogm.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Inherited
public @interface NodeEntity {
    String CLASS = "org.neo4j.ogm.annotation.NodeEntity";
    String LABEL = "label";

    String label() default "";
}
```

Neo4j 是一个图形数据库，所以程序中的实体对象要存入数据库时，将被映射为数据库图形。代码清单 11-7 是这种映射的部分实现代码。它的实现原理是，将实体对象转化为数据库可以识别的查询语句，实现对象到数据的转换。

代码清单 11-7　实体对象映射为数据库图形的部分源代码

```
package org.neo4j.ogm.mapper;
......
public class EntityGraphMapper implements EntityToGraphMapper {
    public CypherContext map(Object entity, int horizon) {
```

```
            if(entity == null) {
                throw new NullPointerException("Cannot map null object");
            } else {
                SingleStatementCypherCompiler compiler = new SingleStatementCypher
Compiler();
                Iterator i$ = this.mappingContext.mappedRelationships().iterator();

                while(i$.hasNext()) {
                    MappedRelationship mappedRelationship = (MappedRelationship)
i$.next();
                    this.logger.debug("context-init: (${})-[:{}]->(${})", new Object
[]{Long.valueOf(mappedRelationship.getStartNodeId()), mappedRelationship.
getRelationshipType(), Long.valueOf(mappedRelationship.getEndNodeId())});
                    compiler.context().registerRelationship(mappedRelationship);
                }

                this.logger.debug("context initialised with {} relationships",
Integer.valueOf(this.mappingContext.mappedRelationships().size()));
                if(this.isRelationshipEntity(entity)) {
                    entity = this.entityAccessStrategy.getStartNodeReader(this.
metaData.classInfo(entity)).read(entity);
                    if(entity == null) {
                        throw new RuntimeException("@StartNode of relationship
entity may not be null");
                    }
                }

                this.mapEntity(entity, horizon, compiler);
                this.deleteObsoleteRelationships(compiler);
                return compiler.compile();
            }
        }
        ......
    }
```

当从 Neo4j 数据库中读取数据时，Neo4j 将数据库中的图形还原为实体对象。代码清单 11-8 是实现这种功能的部分源代码，即将从数据库中查询得到的数据集合转化为实体对象，实现从数据到对象的转换。

代码清单 11-8 数据库中图形还原为实体对象的部分源代码

```
package org.neo4j.ogm.mapper;
......
public class GraphEntityMapper implements GraphToEntityMapper<GraphModel> {
        private void mapNodes(GraphModel graphModel, List<Long> nodeIds) {
```

```
        Iterator i$ = graphModel.getNodes().iterator();

        while(i$.hasNext()) {
            NodeModel node = (NodeModel)i$.next();
            Object entity = this.mappingContext.getNodeEntity(node.getId());

            try {
                if(entity == null) {
                    entity = this.mappingContext.registerNodeEntity(this.entity
Factory.newObject(node), node.getId());
                }

                this.setIdentity(entity, node.getId());
                this.setProperties(node, entity);
                this.mappingContext.remember(entity);
                nodeIds.add(node.getId());
            } catch (BaseClassNotFoundException var7) {
                this.logger.debug(var7.getMessage());
            }
        }

    }
    ......
}
```

11.2.2　持久化实现原理

关系型数据库都使用了 JPA 的一套执行标准，它结合使用 Hibernate 实现了实体的持久化。后续的数据库管理设计都遵循了 JPA 这一个标准规范，提供相同的访问数据库的 API。图 11-1 是 JPA、MongoDB、Neo4j 三种不同的资源库接口定义的相同的继承关系。

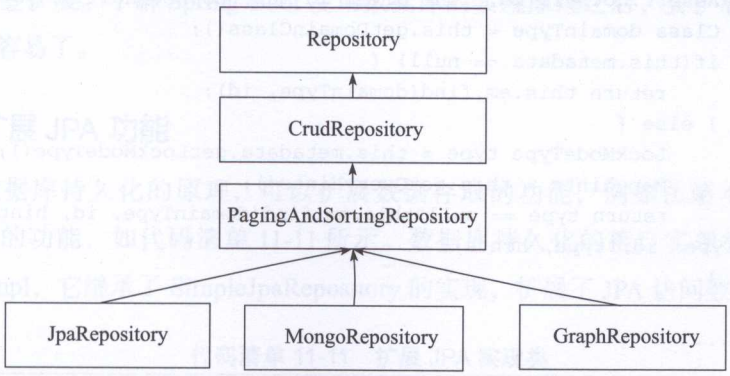


图 11-1　数据库资源库接口的继承关系

这就不难理解，为什么在 Spring Boot 中使用数据库，对于不同种类的数据库，几乎都可以使用相同的方法来访问。但是，上面不同数据库的资源库接口定义虽然有相同的继承关系，它们的实现方法却是不同的，JPA 由 SimpleJpaRepository 实现了 JpaRepository，如代码清单 11-9 所示。

代码清单 11-9 JPA 数据库持久化源代码片段

```
    package org.springframework.data.jpa.repository.support;
    ......
    @Repository
    @Transactional(
        readOnly = true
    )
    public class SimpleJpaRepository<T, ID extends Serializable> implements
JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
        private static final String ID_MUST_NOT_BE_NULL = "The given id must not be null!";
        private final JpaEntityInformation<T, ?> entityInformation;
        private final EntityManager em;
        private final PersistenceProvider provider;
        private CrudMethodMetadata metadata;

        public SimpleJpaRepository(JpaEntityInformation<T, ?> entityInformation, Entity
Manager entityManager) {
            Assert.notNull(entityInformation);
            Assert.notNull(entityManager);
            this.entityInformation = entityInformation;
            this.em = entityManager;
            this.provider = PersistenceProvider.fromEntityManager(entityManager);
        }
        ......
        public T findOne(ID id) {
            Assert.notNull(id, "The given id must not be null!");
            Class domainType = this.getDomainClass();
            if(this.metadata == null) {
                return this.em.find(domainType, id);
            } else {
                LockModeType type = this.metadata.getLockModeType();
                Map hints = this.getQueryHints();
                return type == null?this.em.find(domainType, id, hints):this.em.
find(domainType, id, type, hints);
            }
        }
        ......
    }
```

而对于 Neo4j 来说，它使用 GraphRepositoryImpl 实现了 GraphRepository，如代码清单 11-10 所示。

代码清单 11-10　Neo4j 数据库持久化源代码片段

```
package org.springframework.data.neo4j.repository;
......
@Repository
public class GraphRepositoryImpl<T> implements GraphRepository<T> {
    private static final int DEFAULT_QUERY_DEPTH = 1;
    private final Class<T> clazz;
    private final Session session;

    public GraphRepositoryImpl(Class<T> clazz, Session session) {
        this.clazz = clazz;
        this.session = session;
    }
    ......
    public T findOne(Long id) {
        return this.session.load(this.clazz, id);
    }
    ......
}
```

11.3　扩展数据存取的功能

使用数据库是应用系统最基本的功能需求，同时也是最频繁和最复杂的功能需求。Spring Boot 始终以使用简单为基准，提供了一套以 JPA 的标准规范来设计的数据存取方法，虽然功能相当强大，但往往不能适合一些复杂的功能需求，这就需要对数据存取的功能做一些扩展。了解 Spring Boot 使用数据库的实现原理之后，要扩展数据存取的功能就比较容易了。

11.3.1　扩展 JPA 功能

根据数据库持久化的原理，可以扩展数据存取的功能，例如在第 4 章中，实现了扩展 JPA 的功能，如代码清单 11-11 所示。数据库持久化的接口实现类 ExpandJpaRepositoryImpl，它继承了 SimpleJpaRepository 的实现，扩展了 JPA 访问数据库的功能。

代码清单 11-11　扩展 JPA 实现类

```
public class ExpandJpaRepositoryImpl<T,ID extends Serializable> extends Simple
```

```
JpaRepository<T,ID> implements ExpandJpaRepository<T,ID> {
        private final EntityManager entityManager;
        private final JpaEntityInformation<T, ?> entityInformation;

        public ExpandJpaRepositoryImpl(JpaEntityInformation<T, ?> entityInformation,
 EntityManager entityManager) {
            super(entityInformation, entityManager);
            this.entityManager = entityManager;
            this.entityInformation = entityInformation;
        }
        ......
    }
```

11.3.2 扩展 Neo4j 功能

遵循 JPA 标准规范来设计，这对于新型的 Neo4j 数据库来说是一个挑战。在 JPA 中，可以使用如下定义来执行一个分页的查询：

```
@Query("select t from User t where t.name like :name")
Page<User> findByName(@Param("name") String name, Pageable pageRequest);
```

但是这种方法对于 Neo4j 来说，却会导致严重的错误，如下定义是无法被正常执行的：

```
@Query("MATCH (m:Movie) WHERE m.name =~ ('(?i).*'+{name}+'.*') RETURN m")
Page<Movie> findByName(@Param("name") String name, Pageable pageable);
```

所以为了实现这种分页查询，需要编写一个全局扩展类来实现，如代码清单 11-12 所示，它调用了 Neo4j 的底层实现方法 org.neo4j.ogm.session.Session 来执行分页查询。

代码清单 11-12 Neo4j 分页查询服务类

```
@Service
public class PagesService<T> {
    @Autowired
    private Session session;

    public Page<T> findAll(Class<T> clazz, Pageable pageable, Filters filters){
        Collection data = this.session.loadAll(clazz, filters, convert(pageable.
getSort()), new Pagination(pageable.getPageNumber(), pageable.getPageSize()), 1);
        return updatePage(pageable, new ArrayList(data));
    }

    private Page<T> updatePage(Pageable pageable, List<T> results) {
```

```
            int pageSize = pageable.getPageSize();
            int pageOffset = pageable.getOffset();
            int total = pageOffset + results.size() + (results.size() == pageSize?pageSize:0);
            return new PageImpl(results, pageable, (long)total);
        }

        private SortOrder convert(Sort sort) {
            SortOrder sortOrder = new SortOrder();
            if(sort != null) {
                Iterator var3 = sort.iterator();

                while(var3.hasNext()) {
                    Sort.Order order = (Sort.Order)var3.next();
                    if(order.isAscending()) {
                        sortOrder.add(new String[]{order.getProperty()});
                    } else {
                        sortOrder.add(SortOrder.Direction.DESC, new String[]{order.
getProperty()});
                    }
                }
            }
            return sortOrder;
        }
    }
```

这样在进行分页查询时，就可以调用这个服务类，代码清单 11-13 是使用电影名称，分页查询电影列表的一个实现例子。

代码清单 11-13　使用分页查询服务类进行分页查询的例子

```
    @Autowired
        private PagesService<Movie> pagesService;
        ......
        @RequestMapping(value="/list")
        public Page<Movie> list(HttpServletRequest request) throws Exception{
            String name = request.getParameter("name");
            String page = request.getParameter("page");
            String size = request.getParameter("size");
            Pageable pageable = new PageRequest(page==null? 0: Integer.parseInt(page),
size==null? 10:Integer.parseInt(size),
                    new Sort(Sort.Direction.DESC, "id"));

            Filters filters = new Filters();
            if (!StringUtils.isEmpty(name)) {
                Filter filter = new Filter("name", name);
                filters.add(filter);
```

```
        }
        return pagesService.findAll(Movie.class, pageable, filters);
    }
```

11.4 小结

在 Spring Boot 中访问数据库为什么如此简单？从对一些核心源代码的分析中可知，它始终遵循一套在业界中广为流行的 JPA 标准规范来设计，无论是哪种数据库，它都能使用相同且简单的方法来访问。只是，这对于一些复杂的功能需求来说，未免有些欠缺。所以，在认识它的实现原理之后，需要借助一些底层调用来加强和扩展访问数据库的功能。

通过分析一些核心源代码，我们认识到 Spring Boot 提供的一些可以简单使用的组件中蕴藏的强大功能，也是通过内部的复杂实现来完成的。下一章将剖析微服务的内部实现原理，看看那些可以拿来即用的微服务，其内部的实现原理又是怎样的。

第 12 章 Chapter 12

微服务核心技术实现原理

Spring Cloud 是基于对 Netflix 开源组件进一步封装的一套云应用开发工具，可以用来开发各种微服务应用，它包含很多组件（或子项目），表 12-1 列出了一些主要组件及其功能说明。

表 12-1 Spring Cloud 组件列表

组件	功能
spring-cloud-netflix	集成多种 Netflix 组件提供的开发工具包，包括 Eureka、Hystrix、Zuul、Archaius 等
spring-cloud-eureka	发现服务工具包，用于实现云端的负载均衡和中间层服务器的故障转移等功能
spring-cloud-hystrix	容错和监控管理工具，通过控制服务和第三方库的节点，对延迟和故障提供更强大的容错能力
spring-cloud-zuul	边缘服务工具包，提供动态路由、监控、弹性、安全等的边缘服务
spring-cloud-bus	事件总线工具，用于在集群中使用分布式消息传播状态变化，可与 spring-cloud-config 结合使用，实现配置的在线更新功能
spring-cloud-cli	基于 Spring Boot CLI 的命令行工具，可以快速建立云组件
spring-cloud-config	配置管理开发工具包，可以把配置放到远程服务器上，目前支持本地存储、Git 以及 Subversion
spring-cloud-security	安全管理工具包，为应用程序添加安全控制管理等功能
spring-cloud-cloudfoundry	通过 Oauth2 协议绑定服务到 CloudFoundry，CloudFoundry 是 VMware 推出的开源 PaaS 云平台
spring-cloud-consul	封装了 Consul 操作，Consul 是一个服务发现与配置工具，与 Docker 容器可以无缝集成

（续）

组件	功能
spring-cloud-stream	消息收发工具包，可以在云应用中使用简单的模式与 Redis、RabbitMQ、Kafka 等收发消息
spring-cloud-zookeeper	操作 Zookeeper 的工具包，用于使用 Zookeeper 方式的服务注册和发现

在第 8 章的实例工程中已经使用了其中的配置管理、发现服务、监控服务、动态路由、断路器、负载均衡等功能。本章将从实现的角度探索配置管理、发现服务和负载均衡服务等的实现原理。

12.1 配置管理实现原理

在第 8 章的实例中，我们知道，配置管理的在线更新功能使用事件总线，即 spring-cloud-bus 来发布状态变化，并且使用分布式消息来发布更新事件，而分布式消息最终使用了 RabbitMQ 来实现消息收发。

12.1.1 在线更新流程

使用配置管理，实现在线更新一般遵循下列步骤：

1）更新 Git 仓库的配置文件。

2）以 POST 指令触发更新请求。

3）配置管理服务器从 Git 仓库中读取配置文件，并将配置文件分发给各个客户端，同时在 RabbitMQ 中发布一个更新消息。

4）客户端订阅 RabbitMQ 消息，收到消息后执行更新。

在使用配置管理的演示实例中，使用如下 POST 指令来触发在线更新：

```
curl -X POST http://localhost:8888/bus/refresh
```

接收这个更新指令的实现方法如代码清单 12-1 所示，其中的 publish 将会发布一个更新事件，调用 RabbitMQ 进行消息发布，然后由客户端收到消息后执行更新。代码中定义了请求更新的链接 refresh，并可使用 destination 来指定更新目标。

代码清单 12-1 接收更新指令的源代码

```
package org.springframework.cloud.bus.endpoint;
......
```

```
public class RefreshBusEndpoint extends AbstractBusEndpoint {
    public RefreshBusEndpoint(ApplicationEventPublisher context, String id, Bus
Endpoint delegate) {
        super(context, id, delegate);
    }

    @RequestMapping(
        value = {"refresh"},
        method = {RequestMethod.POST}
    )
    @ResponseBody
    public void refresh(@RequestParam(
    value = "destination",
    required = false
) String destination) {
        this.publish(new RefreshRemoteApplicationEvent(this, this.getInstance
Id(), destination));
    }
}
```

12.1.2　更新消息的分发原理

配置管理服务器中的消息分发是从 spring-cloud-bus 中调用 spring-cloud-stream 组件来实现的，而 spring-cloud-stream 使用 RabbitMQ 实现了分布式消息的分发。

RabbitMQ 的消息服务一般需要创建一个交换机 Exchange 和一个队列 Queue，然后将交换机和队列进行绑定。而在设计配置服务器时并没有做这方面的工作，所做的工作仅仅是配置引用 spring-cloud-bus 的依赖和设置连接 RabbitMQ 服务器的参数而已。这个工作其实已经由 spring-cloud-stream 帮我们实现了。

从 RabbitMessageChannelBinder 的源代码中可以看到这部分的实现原理，代码清单 12-2 是一个消息发布方的队列绑定的实现。其中 exchangeName 是一个交换机的名字，baseQueueName 是一个队列的名字，并且从代码中也可以看出它使用了 TopicExchange 交换机，这是 RabbitMQ 中 4 种交换机（Fanout、Direct、Topic、Header）的其中一种，并且也可以看出代码中使用 setRoutingKey 将交换机和队列做了绑定。

代码清单 12-2　RabbitMessageChannelBinder 的源代码

```
package org.springframework.cloud.stream.binder.rabbit;
......
public class RabbitMessageChannelBinder extends MessageChannelBinderSupport
implements DisposableBean {
```

```
        private AmqpOutboundEndpoint buildOutboundEndpoint(String name, RabbitMessage
ChannelBinder.RabbitPropertiesAccessor properties, RabbitTemplate rabbitTemplate) {
            String prefix = properties.getPrefix(this.defaultPrefix);
            String exchangeName = applyPrefix(prefix, name);
            String partitionKeyExtractorClass = properties.getPartitionKeyExtractor
Class();
            Expression partitionKeyExpression = properties.getPartitionKeyExpression();
            TopicExchange exchange = new TopicExchange(exchangeName);
            this.declareExchange(exchangeName, exchange);
            AmqpOutboundEndpoint endpoint = new AmqpOutboundEndpoint(rabbitTemplate);
            endpoint.setExchangeName(exchange.getName());
            String baseQueueName = exchangeName + ".default";
            if(partitionKeyExpression == null && !StringUtils.hasText(partitionKey
ExtractorClass)) {
                Queue var15 = new Queue(baseQueueName, true, false, false, this.
queueArgs(properties, baseQueueName));
                this.declareQueue(baseQueueName, var15);
                this.autoBindDLQ(baseQueueName, baseQueueName, properties);
                endpoint.setRoutingKey(name);
                org.springframework.amqp.core.Binding var16 = BindingBuilder.bind
(var15).to(exchange).with(name);
                this.declareBinding(baseQueueName, var16);
            } else {
                endpoint.setExpressionRoutingKey(EXPRESSION_PARSER.parseExpression
(this.buildPartitionRoutingExpression(name)));

                for(int i = 0; i < properties.getNextModuleCount(); ++i) {
                    String partitionSuffix = "-" + i;
                    String partitionQueueName = baseQueueName + partitionSuffix;
                    Queue queue = new Queue(partitionQueueName, true, false, false,
 this.queueArgs(properties, partitionQueueName));
                    this.declareQueue(queue.getName(), queue);
                    this.autoBindDLQ(baseQueueName, baseQueueName + partitionSuffix,
 properties);
                    this.declareBinding(queue.getName(), BindingBuilder.bind(queue).
to(exchange).with(name + partitionSuffix));
                }
            }

            this.configureOutboundHandler(endpoint, properties);
            return endpoint;
        }
        ......
    }
```

现在我们更加明白，为什么使用Spring Boot可以那么简单，就是因为一些复

杂的配置和方法都已经由 Spring Boot 及其所调用的一些组件实现了。至于在使用 RabbitMQ 中进行消息发布的实现，最终是由 RabbitTemplate 执行 doSend，将消息发布到 RabbitMQ 服务器上，如代码清单 12-3 所示。

代码清单 12-3　消息发布源代码

```
    package org.springframework.amqp.rabbit.core;
    ......
    public class RabbitTemplate extends RabbitAccessor implements BeanFactoryAware,
RabbitOperations, MessageListener, ListenerContainerAware, Listener {
        public void send(Message message) throws AmqpException {
            this.send(this.exchange, this.routingKey, message);
        }

        public void send(final String exchange, final String routingKey, final Message
 message, final CorrelationData correlationData) throws AmqpException {
            this.execute(new ChannelCallback() {
                public Object doInRabbit(Channel channel) throws Exception {
                    RabbitTemplate.this.doSend(channel, exchange, routingKey,
message,RabbitTemplate.this.returnCallback != null && ((Boolean)RabbitTemplate.
this.mandatoryExpression.getValue(RabbitTemplate.this.evaluationContext,
message, Boolean.class)).booleanValue(), correlationData);
                    return null;
                }
            }, this.obtainTargetConnectionFactoryIfNecessary(this.sendConnection
FactorySelectorExpression, message));
        }
        ......
    }
```

使用配置管理服务的客户端都订阅了 RabbitMQ 服务器的消息，当收到更新消息时，即从配置管理服务器中取得更新文件，然后在本地上执行更新配置的流程。

有关消息的发布和订阅的实现方法，最后通过一个简单的实例，使用 spring-cloud-stream，更加形象地说明这种分布式消息的发布和接收的原理。

12.2　发现服务源代码剖析

使用发现服务时，只要简单地通过注解 @ EnableEurekaServer 来标注一个应用为发现服务器，通过注解 @EnableDiscoveryClient 来标注一个应用为发现服务的客户端，服务器就会实现服务注册的功能，客户端将会从服务器中取得已经注册的可用服务列表。

12.2.1 服务端的服务注册功能

服务注册是发现服务的一个主要功能，可以从注解 @EnableEurekaServer 的定义中顺藤摸瓜地找到其实现的源代码，如代码清单 12-4 所示，其中 @Import 将导入一个发现服务的配置：EurekaServerConfiguration。

代码清单 12-4 注解 @EnableEurekaServer 源代码

```
package org.springframework.cloud.netflix.eureka.server;
......
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import({EurekaServerConfiguration.class})
public @interface EnableEurekaServer {
}
```

通过 EurekaServerConfiguration，又引入了一些配置，如增加了监控器和过滤器的配置等，其中一个 InstanceRegistry 的配置将实现对客户端的注册，如代码清单 12-5 所示。

代码清单 12-5 EurekaServerConfiguration 源代码

```
package org.springframework.cloud.netflix.eureka.server;
......
@Configuration
@Import({EurekaServerInitializerConfiguration.class})
@EnableDiscoveryClient
@EnableConfigurationProperties({EurekaDashboardProperties.class})
public class EurekaServerConfiguration extends WebMvcConfigurerAdapter {
    ......
    @Bean
    public PeerAwareInstanceRegistry peerAwareInstanceRegistry(ServerCodecs
serverCodecs) {
        this.eurekaClient.getApplications();
        return new InstanceRegistry(this.eurekaServerConfig, this.eurekaClient
Config,serverCodecs, this.eurekaClient, this.expectedNumberOfRenewsPerMin,
this.defaultOpenForTrafficCount);
    }
    ......
}
```

上面的 InstanceRegistry 调用了超类 AbstractInstanceRegistry 的 register 对客户端进

行注册，然后将在线的客户端存入注册队列 recentRegisteredQueue 中，如代码清单 12-6 所示。

代码清单 12-6　超类 AbstractInstanceRegistry 源代码片段

```
package com.netflix.eureka.registry;
.....
public abstract class AbstractInstanceRegistry implements InstanceRegistry {
    private static final Logger logger = LoggerFactory.getLogger(AbstractInstance
Registry.class);
    private static final String[] EMPTY_STR_ARRAY = new String[0];
    private final ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>
 registry = new ConcurrentHashMap();
    protected Map<String, RemoteRegionRegistry> regionNameVSRemoteRegistry = new
HashMap();
    protected final ConcurrentMap<String, InstanceStatus> overriddenInstance
StatusMap;
    private final AbstractInstanceRegistry.CircularQueue<Pair<Long, String>>
 recentRegisteredQueue;
    private final AbstractInstanceRegistry.CircularQueue<Pair<Long, String>>
 recentCanceledQueue;
    private ConcurrentLinkedQueue<AbstractInstanceRegistry.RecentlyChangedItem>
 recentlyChangedQueue;
    .....
    public void register(InstanceInfo r, int leaseDuration, boolean isReplication) {
        try {
            this.read.lock();
            Object gMap = (Map)this.registry.get(r.getAppName());
            EurekaMonitors.REGISTER.increment(isReplication);
            if(gMap == null) {
                ConcurrentHashMap existingLease = new ConcurrentHashMap();
                gMap = (Map)this.registry.putIfAbsent(r.getAppName(), existing
Lease);
                if(gMap == null) {
                    gMap = existingLease;
                }
            }
        .....
      ((Map)gMap).put(r.getId(), lease2);
     AbstractInstanceRegistry.CircularQueue overriddenStatusFromMap1 = this.
recentRegisteredQueue;
    synchronized(this.recentRegisteredQueue) {
            this.recentRegisteredQueue.add(new Pair(Long.valueOf(System.
currentTimeMillis()), r.getAppName() + "(" + r.getId() + ")"));
            }
      .....
```

```
        }
    }
```

客户端在发现服务器中注册之后，就可以在发现服务器的控制台上，查看在线的客户端列表。这里有一个缺陷，如果客户端关闭了，在发现服务器的控制台中，还能查到这个客户端，只有重启发现服务器，才能更新这个客户端列表。

12.2.2 客户端注册和提取服务列表

客户端除了自身在发现服务器上注册之外，它还要从服务器中取得已经注册的其他客户端，以得到一个可用的服务列表（其他注册的客户端）。这部分的核心源代码可以在 com.netflix.discovery.DiscoveryClient 中找到。

客户端自身注册的实现方法如代码清单 12-7 所示。这里，主要是将客户端的名称、IP 地址和端口等信息通过一个 instanceInfo 对象发给发现服务器进行注册。

代码清单 12-7 客户端注册源代码

```
package com.netflix.discovery;
......
public class DiscoveryClient implements EurekaClient {
......
    boolean register() throws Throwable {
        logger.info("DiscoveryClient_" + this.appPathIdentifier + ": registering
 service...");
        if(this.shouldUseExperimentalTransportForRegistration()) {
            EurekaHttpResponse response1;
            try {
                response1 = this.eurekaTransport.registrationClient.register
(this.instanceInfo);
            } catch (Exception var7) {
                logger.warn("{} - registration failed {}", new Object[]{"Discovery
Client_" + this.appPathIdentifier, var7.getMessage(), var7});
                throw var7;
            }

            this.isRegisteredWithDiscovery = true;
            if(logger.isInfoEnabled()) {
                logger.info("{} - registration status: {}", "DiscoveryClient_"
 + this.appPathIdentifier, Integer.valueOf(response1.getStatusCode()));
            }

            return response1.getStatusCode() == 204;
        } else {
```

```
                ClientResponse response = null;

                boolean e;
                try {
                    response = this.makeRemoteCall(DiscoveryClient.Action.Register);
                    this.isRegisteredWithDiscovery = true;
                    logger.info("{} - registration status: {}", "DiscoveryClient_"
 + this.appPathIdentifier, response != null?Integer.valueOf(response.
getStatus()):"not sent");
                    e = response != null && response.getStatus() == 204;
                } catch (Throwable var8) {
                    logger.warn("{} - registration failed {}", new Object[]{"Disco
very | Client_" + this.appPathIdentifier, var8.getMessage(), var8});
                    throw var8;
                } finally {
                    this.closeResponse(response);
                }

                return e;
            }
        }
        ......
    }
```

客户端执行注册使用计划任务的方式来实现，而客户端从发现服务器中更新其他在线的客户端列表，也使用了一个定时任务来管理。代码清单 12-8 使用一个定时任务 TimerTask 定时从发现服务器中取得其他在线的客户端列表，以备使用。

代码清单 12-8　更新客户端列表定时器源代码

```
package com.netflix.discovery;
......
public class DiscoveryClient implements EurekaClient {
......
    private TimerTask getServiceUrlUpdateTask(final String zone) {
        return new TimerTask() {
            public void run() {
                try {
                    List e = DiscoveryClient.this.timedGetDiscoveryServiceUrls
(zone);
                    if(e.isEmpty()) {
                        DiscoveryClient.logger.warn("The service url list is
 empty");
                        return;
                    }
```

```
                    if(!e.equals(DiscoveryClient.this.eurekaServiceUrls.get
())) {
                        DiscoveryClient.logger.info("Updating the serviceUrls
 as they seem to have changed from {} to {} ", Arrays.toString(((List)DiscoveryClient.
this.eurekaServiceUrls.get()).toArray()), Arrays.toString(e.toArray()));
                        DiscoveryClient.this.eurekaServiceUrls.set(e);
                    }
                } catch (Throwable var2) {
                    DiscoveryClient.logger.error("Cannot get the eureka service
 urls :", var2);
                }
            }
        };
    }
    ......
    }
```

单纯的发现服务，并不能看出它有多大的用途，它只有与动态路由、负载均衡和监控服务等一起使用，才能发挥其强大的功能。

12.3 负载均衡源代码剖析

当一个应用启用发现服务的功能之后，会默认启用 Ribbon 的负载均衡服务。Ribbon 通过发现服务获取在线的客户端，为具有多个实例的客户端建立起负载均衡实例列表，然后通过一定的负载均衡算法，实现负载均衡的管理机制。

如代码清单 12-9 所示，Ribbon 默认结合使用 Eureka 发现服务，启用负载均衡管理机制。当没有配置“ribbon.eureka.enabled”参数时，它默认被设定为 true。

代码清单 12-9 读取启用负载均衡配置的源代码

```
package org.springframework.cloud.netflix.ribbon.eureka;
......
@Configuration
@EnableConfigurationProperties
@ConditionalOnClass({DiscoveryEnabledNIWSServerList.class})
@ConditionalOnBean({SpringClientFactory.class})
@ConditionalOnProperty(
    value = {"ribbon.eureka.enabled"},
    matchIfMissing = true
)
```

```
@AutoConfigureAfter({RibbonAutoConfiguration.class})
@RibbonClients(
    defaultConfiguration = {EurekaRibbonClientConfiguration.class}
)
public class RibbonEurekaAutoConfiguration {
    public RibbonEurekaAutoConfiguration() {
    }
}
```

看看负载均衡服务是如何进行初始化的，就更清楚它的实现原理了。代码清单 12-10 是 BaseLoadBalancer 的部分源代码。这里，程序加载了一些初始配置，如可用的负载均衡服务实例列表、监控计数和服务状态监听等，其中一个重要的设置，即设定默认的负载均衡规则 RoundRobinRule，这是一个使用简单轮询算法的负载均衡规则。一个负载均衡服务的实现，就是通过一定的负载均衡算法，从可用的服务实例列表中，为请求者提供一个可用的服务。

代码清单 12-10　BaseLoadBalancer 源代码片段

```
package com.netflix.loadbalancer;
......
public class BaseLoadBalancer extends AbstractLoadBalancer implements Prime
ConnectionListener, IClientConfigAware {
    private static Logger logger = LoggerFactory.getLogger(BaseLoadBalancer.
class);
    private static final IRule DEFAULT_RULE = new RoundRobinRule();
    private static final String DEFAULT_NAME = "default";
    private static final String PREFIX = "LoadBalancer_";
    protected IRule rule;
    protected IPing ping;
    @Monitor(
        name = "LoadBalancer_AllServerList",
        type = DataSourceType.INFORMATIONAL
    )
    protected volatile List<Server> allServerList;
    @Monitor(
        name = "LoadBalancer_UpServerList",
        type = DataSourceType.INFORMATIONAL
    )
    public BaseLoadBalancer() {
        this.rule = DEFAULT_RULE;
        this.ping = null;
        this.allServerList = Collections.synchronizedList(new ArrayList());
        this.upServerList = Collections.synchronizedList(new ArrayList());
        this.allServerLock = new ReentrantReadWriteLock();
```

```
        this.upServerLock = new ReentrantReadWriteLock();
        this.name = "default";
        this.lbTimer = null;
        this.pingIntervalSeconds = 10;
        this.maxTotalPingTimeSeconds = 5;
        this.serverComparator = new ServerComparator();
        this.pingInProgress = new AtomicBoolean(false);
        this.counter = Monitors.newCounter("LoadBalancer_ChooseServer");
        this.enablePrimingConnections = false;
        this.changeListeners = new CopyOnWriteArrayList();
        this.serverStatusListeners = new CopyOnWriteArrayList();
        this.name = "default";
        this.ping = null;
        this.setRule(DEFAULT_RULE);
        this.setupPingTask();
        this.lbStats = new LoadBalancerStats("default");
    }
    ......
}
```

再来看看 RoundRobinRule 的实现代码，如代码清单 12-11 所示。从中可以看出，它使用一个循环，从可用的服务列表中，按顺序选择一个可用的服务。其中，一个选择服务的请求不能超过 10 次无效的尝试，这仅仅是一个循环语句的安全设计而已，并不会影响一次选择查询。

从整体上来说，使用哪种负载均衡算法，对于整个负载均衡服务来说，影响并不是很大。除了 RoundRobinRule，Ribbon 还提供了其他一些负载均衡规则，如加权响应时间规则 WeightedResponseTimeRule、区域感知规则 ZoneAvoidanceRule、随机规则 RandomRule 等。

代码清单 12-11 RoundRobinRule 源代码片段

```
package com.netflix.loadbalancer;
......
public class RoundRobinRule extends AbstractLoadBalancerRule {
AtomicInteger nextIndexAI;
    private static Logger log = LoggerFactory.getLogger(RoundRobinRule.class);
    static final boolean availableOnly = false;

    public RoundRobinRule() {
        this.nextIndexAI = new AtomicInteger(0);
    }
......
    public Server choose(ILoadBalancer lb, Object key) {
```

```
        if(lb == null) {
            log.warn("no load balancer");
            return null;
        } else {
            Server server = null;
            boolean index = false;
            int count = 0;

            while(true) {
                if(server == null && count++ < 10) {
                    List upList = lb.getServerList(true);
                    List allList = lb.getServerList(false);
                    int upCount = upList.size();
                    int serverCount = allList.size();
                    if(upCount != 0 && serverCount != 0) {
                        int var10 = this.nextIndexAI.incrementAndGet() % server
Count;

                        server = (Server)allList.get(var10);
                        if(server == null) {
                            Thread.yield();
                        } else {
                            if(server.isAlive() && server.isReadyToServe()) {
                                return server;
                            }

                            server = null;
                        }
                        continue;
                    }

                    log.warn("No up servers available from load balancer: " + lb);
                    return null;
                }

                if(count >= 10) {
                    log.warn("No available alive servers after 10 tries from
load balancer: " + lb);
                }

                return server;
            }
        }
    }
    ......
}
```

例如在第 9 章的演示中，当把其中的 data 服务配置为两个或两个以上的运行实例时，就会启动 Ribbon 的负载均衡机制，用来管理其他服务对 data 服务的访问。我们只是使用了默认的简单轮询负载均衡规则，即第一次调用将访问第一个服务实例，第二次调用将访问第二个服务实例，以此类推，当调用到服务列表的最后一个服务后再从头来过。

12.4 分布式消息实现原理演示

使用 RabbitMQ 实现分布式消息分发，在分布式系统中具有很大的用途，为各个应用之间传递消息和数据提供了很大的方便，并且其松散耦合的结构和异步处理的机制不会影响系统的性能。

使用 spring-cloud-stream 可以非常简单地使用 RabbitMQ 的异步消息，Spring Cloud 的配置管理中的分布式消息分发也是通过调用 spring-cloud-stream 组件来实现的。下面将创建一个消息生产者和一个消息消费者来演示消息分发的实现原理。

12.4.1 消息生产者

消息生产者的实现如代码清单 12-12 所示，这里主要创建了一个 POST 接口“/send”，以接收传入的 Map 对象作为参数，使用 MessageChannel 的 send 方法，将消息发布到 RabbitMQ 的消息队列上。使用 Map 对象的目的，是保证消息生产者和消息消费者之间，可以使用相同的对象来存取消息的内容，这就要求双方必须事先约定 Map 对象的字段。

代码清单 12-12 消息生产者主程序

```
@EnableBinding(Source.class)
@RestController
@SpringBootApplication
public class SenderApplication {
    @Autowired
    @Output(Source.OUTPUT)
    private MessageChannel channel;

    @RequestMapping(method = RequestMethod.POST, path = "/send")
    public void write (@RequestBody Map<String, Object> msg){
        channel.send(MessageBuilder.withPayload(msg).build());
```

```
    }

    public static void main(String[] args) {
        SpringApplication.run(SenderApplication.class, args);
    }
}
```

代码清单 12-13 是消息生产者这个工程的配置文件，其中 stream 配置的目的地为 cloud-stream，客户端订阅时必须使用相同的目的地，注意这里绑定的方式是 output，rabbitmq 是连接消息服务器的一些参数配置。

代码清单 12-13　消息生产者工程配置文件

```
server:
    port: 80
spring:
    cloud:
        stream:
            bindings:
            output:
                destination: cloud-stream
    rabbitmq:
        addresses: amqp://192.168.1.216:5672
        username: alan
        password: alan
```

12.4.2　消息消费者

消息消费者的实现如代码清单 12-14 所示，这里从 SubscribableChannel（即 Sink.INPUT）订阅了通道的消息，它相当于一个监听器，当订阅的通道上有消息发布时，就将消息取回来，然后简单地在控制台上打印出来。这里使用的 Map 对象，跟消息生产者约定了两个使用字段，即 msg 和 name。

代码清单 12-14　消息消费者主程序

```
@EnableBinding(Sink.class)
@IntegrationComponentScan
@MessageEndpoint
@SpringBootApplication
public class ReceiverApplication {
    @ServiceActivator(inputChannel=Sink.INPUT)
    public void accept(Map<String, Object> msg){
        System.out.println(msg.get("msg").toString() + ":" + msg.get("name"));
```

```
    }
    public static void main(String[] args) {
        SpringApplication.run(ReceiverApplication.class, args);
    }
}
```

代码清单 12-15 是消息消费者工程的配置文件，它与消息生产者有相同的目的地，不同的是它的绑定方式是 input，其中 rabbitmq 的配置是相同的，只要连上 RabbitMQ 消息服务器即可。

代码清单 12-15 消息消费者工程配置文件

```
server:
    port: 81
spring:
    cloud:
        stream:
            bindings:
            input:
                destination: cloud-stream
                group: group1
                consumer:
                  durableSubscription: true
    rabbitmq:
        addresses: amqp://192.168.1.216:5672
        username: alan
        password: alan
```

这样，当两个工程都启动之后，输入下列 POST 指令，就可以将消息从消息生产者中发布出去。

```
curl -l -H "Content-type:application/json" -X POST -d'{"msg":"Hello","name
":"RabbitMQ"}' http://localhost/send
```

其中消息结构中的 msg 和 name 就是约定好的字段。这时，在消息消费者的控制台上可以看到接收到了如下信息：

```
Hello:RabbitMQ
```

从上面的演示可以看出，使用 spring-cloud-stream 来实现分布式消息的分发和接收都是非常简单的。上面实例工程的完整代码可以从 GitHub 中检出：https://github.com/chenfromsz/spring-cloud-stream-demo.git.

12.5　小结

本章分析了 Spring Cloud 一系列微服务中配置服务、发现服务和负载均衡服务的实现原理和部分核心源代码，并使用一个简单的实例，演示了使用分布式消息的简单实现方法，从中让我们更加清楚地认识到，Spring Boot 及其一些相关的组件，已经尽量把一些可以实现和做到的功能，都帮我们实现了。所以，虽然使用 Spring Boot 及其相关组件看起来非常简单，但实际上可以实现无比强大的功能，这就是 Spring Boot 及其组件的神奇所在。

有关 Spring Cloud 的更多内容和参考，可以访问其官方网站：http://projects.spring.io/spring-cloud/ 和 http://projects.spring.io/spring-cloud/spring-cloud.html。

Appendix A

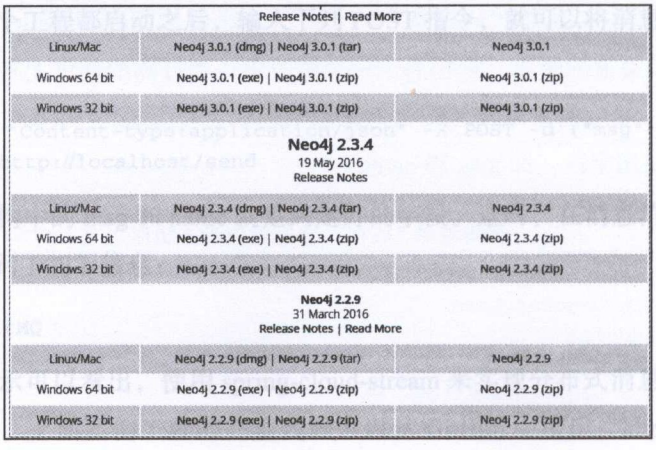
附录 A

安装 Neo4j

Neo4j 数据库有两个版本：社区版和商业版，社区版是开源并且免费的。社区版与商业版功能上没有什么区别，不同的是社区版只能单机使用，商业版可以做分布式集群。单机版最大可以存储 10 亿个节点。

Neo4j 针对不现的操作系统，如 MAC OSX、Linux、Windows 等提供不同的安装包，可以使用下列链接选择下载社区版的不同安装包，打开链接后如图 A-1 所示。

```
http://neo4j.com/download/other-releases/
```



Release Notes | Read More		
Linux/Mac	Neo4j 3.0.1 (dmg) | Neo4j 3.0.1 (tar)	Neo4j 3.0.1
Windows 64 bit	Neo4j 3.0.1 (exe) | Neo4j 3.0.1 (zip)	Neo4j 3.0.1 (zip)
Windows 32 bit	Neo4j 3.0.1 (exe) | Neo4j 3.0.1 (zip)	Neo4j 3.0.1 (zip)
Neo4j 2.3.4 19 May 2016 Release Notes		
Linux/Mac	Neo4j 2.3.4 (dmg) | Neo4j 2.3.4 (tar)	Neo4j 2.3.4
Windows 64 bit	Neo4j 2.3.4 (exe) | Neo4j 2.3.4 (zip)	Neo4j 2.3.4 (zip)
Windows 32 bit	Neo4j 2.3.4 (exe) | Neo4j 2.3.4 (zip)	Neo4j 2.3.4 (zip)
Neo4j 2.2.9 31 March 2016 Release Notes | Read More		
Linux/Mac	Neo4j 2.2.9 (dmg) | Neo4j 2.2.9 (tar)	Neo4j 2.2.9
Windows 64 bit	Neo4j 2.2.9 (exe) | Neo4j 2.2.9 (zip)	Neo4j 2.2.9 (zip)
Windows 32 bit	Neo4j 2.2.9 (exe) | Neo4j 2.2.9 (zip)	Neo4j 2.2.9 (zip)

图 A-1　Neo4j 下载选择

因为本书实例使用的版本是 Neo4j2.3.2，所以选择 Neo4j2.3.4 这个比较接近的版本的安装包下载。对于系统的最低要求，参照官方说明如下：

- CPU：Intel Core i3
- 内存：2GB
- 硬盘：10GB SATA

如果使用 Linux 系统安装，可以先创建一个目录，或者在“/opt”目录中下载 tar 安装包，然后使用下列指令解压即完成安装：

```
#tar -xf 文件名
```

然后切换到 Neo4j 主目录，执行下列指令启动 Neo4j 服务：

```
#./bin/neo4j start
```

如果在 Windows 上安装，使用 Windows 的安装文件，按默认选项执行安装。安装完成后，在程序菜单中打开 Neo4j Community Edition 窗口，如图 A-2 所示。

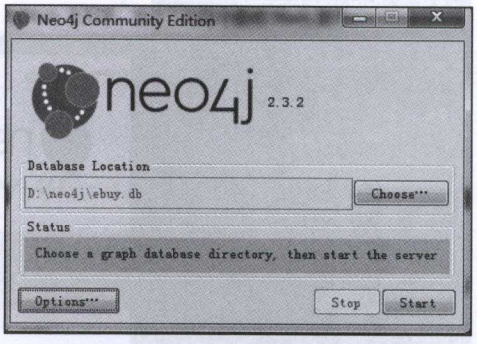


图 A-2　Neo4j 启动窗口

数据库保存位置使用默认的位置即可，单击 Start 按钮，启动 Neo4j 服务。

如果不是在本地调用，需要开启远程调用功能，可以修改 neo4j-server.properties 配置文件来实现，即将下列配置项的注释“#”去掉即可。

```
#org.neo4j.server.webserver.address=0.0.0.0
```

还有一个配置文件 neo4j.properties，可以用来配置使用内存的大小和日志保留时间等参数，这些使用默认的配置即可。

启动 Neo4j 服务之后，可以使用浏览器打开数据库的控制台，假设数据库安装在本地上，使用 URL 就可以打开控制台：

```
http://localhost:7474
```

打开后需要输入默认的用户名和密码：neo4j，首次使用会要求更改初始密码，如图 A-3 所示。

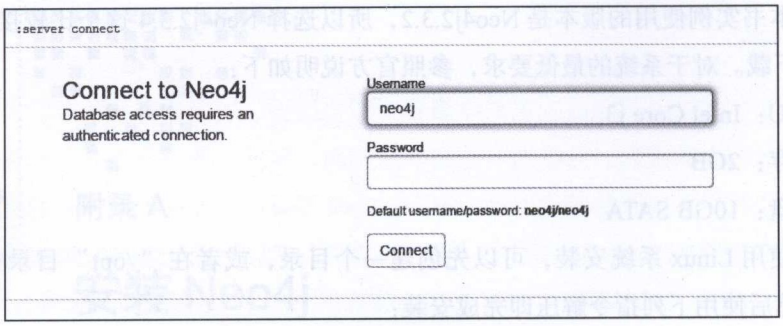


图 A-3 Neo4j 控制台登录界面

登录之后，在控制上可以展开左边侧边栏，如单击 Overview，可以打开操作数据库的面板，查看节点和关系，如图 A-4 所示。

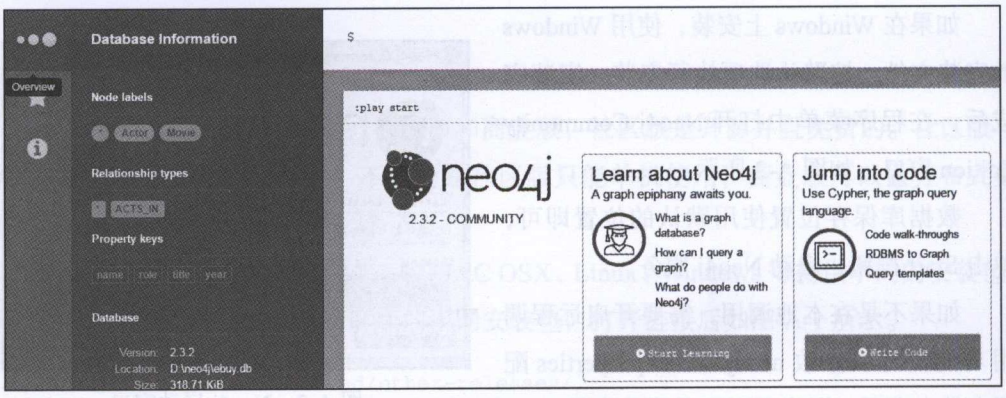


图 A-4 Neo4j 控制台操作面板

Neo4j 控制台是一个用 HTML5 设计的漂亮的操作界面。现在就可以开始使用具有服务器的 Neo4j 图形数据库了。

附录 B Appendix B

安装 MongoDB

MongoDB 提供有 Windows、Linux、OSX、Solaris 等操作系统的安装包，打开如下网址，可以看到如图 B-1 所示的下载选项。

https://www.mongodb.com/download-center?jmp=nav#community

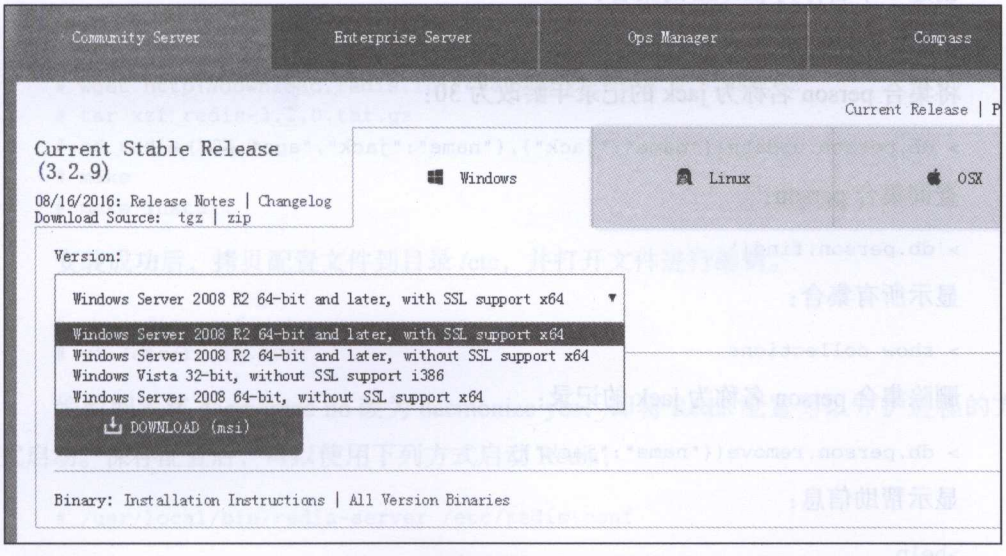


图 B-1 下载 MongoDB 安装包

选择使用 Windows 的安装包，可以选择 All Version Binaries 打开一个新窗口，选择

适合操作系统版本和位数的安装包下载。

例如，在 D：盘中安装了 64 位的 MongoDB 3.2，找到类似如下的 bin 路径，将其配置在 Windows 的环境变量 path 中。

```
D:\Program Files\MongoDB\Server\3.2\bin
```

创建一个保存数据库的目录，如 d:\mongodb\test，然后打开一个命令行窗口，输入如下命令启动 MongoDB 服务：

```
mongod --dbpath=d:\mongodb\test
```

可以看到最后一行输出如下信息，表明服务启动成功，并且开放了访问端口为 27027。

```
…waiting for connections on port 27017
```

打开另一个命令行窗口作为客户端，输入命令 mongo 即可连接上服务器。这样可以使用一些指令做一些简单的操作测试。

显示所有数据库：

```
>show dbs
```

切换到 test 数据库：

```
>use test
```

创建一个名称为 person 的集合：

```
> db.person.insert({"name":"jack","age":25})
```

将集合 person 名称为 jack 的记录年龄改为 30：

```
> db.person.update({"name":"jack"},{"name":"jack","age":30})
```

查询集合 person：

```
> db.person.find()
```

显示所有集合：

```
> show collections
```

删除集合 person 名称为 jack 的记录：

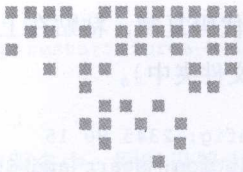
```
> db.person.remove({"name":"jack"})
```

显示帮助信息：

```
>help
```

退出：

```
>exit
```



附录 C Appendix C

安装 Redis

以下安装在 CentOS6.5 上进行。

为了方便在本机上进行测试，可以先安装 tcl 支持环境。

```
# yum install tcl
```

使用下列指令执行下载、解压、安装：

```
# wget http://download.redis.io/releases/redis-3.2.0.tar.gz
# tar xzf redis-3.2.0.tar.gz
# cd redis-3.2.0
# make
# make install
```

安装成功后，拷贝配置文件到目录 /etc，并打开文件进行编辑。

```
# cp redis.conf /etc
# vi /etc/redis.conf
```

编辑配置将 daemonize no 改为 daemonize·yes，即将 Redis 配置为以守护进程的方式启动。保存配置后，可以使用下列方式启动 Redis：

```
# /usr/local/bin/redis-server /etc/redis.conf
```

但是为了更加方便地启动，可以创建一个启动文件，然后将它加入系统服务中：

```
# vi /etc/init.d/redis
```

将下列代码复制，粘贴在上面的编辑中（下列代码保存在第2章实例工程的redis模块的doc文件夹中）。

```
#chkconfig: 2345 80 15
#description: Start and Stop redis
PATH=/usr/local/bin:/sbin:/usr/bin:/bin
REDISPORT=6379
EXEC=/usr/local/bin/redis-server
REDIS_CLI=/usr/local/bin/redis-cli
PIDFILE=/var/run/redis.pid
CONF="/etc/redis.conf"

case "$1" in
    start)
        if [ -f $PIDFILE ]
        then
                echo "$PIDFILE exists, process is already running or crashed"
        else
                echo "Starting Redis server..."
                $EXEC $CONF
        fi
        if [ "$?"="0" ]
        then
              echo "Redis is running..."
        fi
        ;;
    stop)
        if [ ! -f $PIDFILE ]
        then
                echo "$PIDFILE does not exist, process is not running"
        else
                PID=$(cat $PIDFILE)
                echo "Stopping ..."
                $REDIS_CLI -p $REDISPORT SHUTDOWN
                while [ -x ${PIDFILE} ]
               do
                    echo "Waiting for Redis to shutdown ..."
                    sleep 1
                done
                echo "Redis stopped"
        fi
        ;;
  restart|force-reload)
        ${0} stop
        ${0} start
        ;;
```

```
    *)
      echo "Usage: /etc/init.d/redis {start|stop|restart|force-reload}" >&2
          exit 1
esac
```

上面文件保存后，更改其执行权限，将其加入系统服务中，同时设置为自动启动。

```
# chmod +x /etc/init.d/redis
# chkconfig --add redis
# chkconfig redis on
```

使用下列指令查看一下，如果2、3、4、5项为开启状态，即表示配置成功。

```
# chkconfig --list redis
```

现在可以使用下列指令启动Redis服务。

```
# service redis start
```

启动后可使用下列指令在本地测试：

```
# redis-cli
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> get foo
"bar"
127.0.0.1:6379>
127.0.0.1:6379> quit
```

上面测试表示Redis已经正常运行，并开启了默认端口为6379。如果系统开启了防火墙，可以使用下列指令开放6379端口：

```
# vi /etc/sysconfig/iptables
```

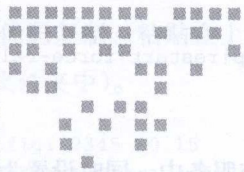
插入一条配置：

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT
```

保存后重启防火墙：

```
# service iptables restart
```

更多信息可以参考Redis的官方网站：http://redis.io/download



Appendix D 附录 D

安装 RabbitMQ

下列安装步骤在 CentOS6.5 上进行。

1. 安装 erlang 语言环境

安装依赖文件：

```
# yum install ncurses-devel
```

下载 erlnag：

```
# wget http://www.erlang.org/download/otp_src_R16B03.tar.gz
```

解压：

```
# tar zxvf otp_src_R16B03.tar.gz
```

安装：

```
# cd otp_src_R16B03
# ./configure
# make
# make install
```

测试执行 erl：

```
# erl
```

即将打印出类似如下的信息：

```
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [async-threads:10] [hipe] [kernel-
poll:false]

Eshell V5.10.4  (abort with ^G)
```

退出 erl：

```
1> halt().
```

2. 安装 Python

如果使用免编译安装包来安装，则忽略此步骤，直接进入第 4 步。

查看原来系统自带的 Python 版本：

```
# python -V
```

如果版本比 2.7 还低，则下载 2.7 的版本，否则跳过此步骤，直接进入第 3 步。

```
# wget http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tgz
```

解压：

```
# tar zxvf  Python-2.7.6.tgz
```

安装：

```
# cd Python-2.7.6
# ./configure
# make && make install
```

废弃原来的 Python，替换为 2.7。

```
# mv /usr/bin/python /usr/bin/python2.4.3
# ln -s /usr/local/bin/python2.7 /usr/bin/python
```

现在再查看版本：

```
# python -V
```

3. 安装 RabbitMQ

安装依赖文件：

```
# yum install xmlto
```

下载：

```
# wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.5.4/rabbitmq-server
```

```
-3.5.4.tar.gz
```

解压：

```
# tar xvzf rabbitmq-server-3.5.4.tar.gz
```

安装：

```
# cd rabbitmq-server-3.5.4
# make
# make install TARGET_DIR=/usr/rabbitmq SBIN_DIR=/usr/rabbitmq/sbin MAN_DIR=/
usr/rabbitmq/man DOC_INSTALL_DIR=/usr/rabbitmq/doc
```

配置环境变量：

```
# vi /etc/profile
```

增加一行：

```
export PATH=$PATH:/usr/rabbitmq/sbin
```

保存后，使用下列指令让配置立即生效：

```
# source /etc/profile
```

启用 management plugin：

```
# mkdir /etc/rabbitmq
# rabbitmq-plugins enable rabbitmq_management
```

启动 RabbitMQ 服务：

```
# rabbitmq-server -detached
```

如果要停止 RabbitMQ 服务，则可以使用如下指令：

```
# rabbitmqctl stop
```

4. 使用免编译安装包安装

如果已经使用第 3 步的方法安装成功，则忽略此步骤。

下载免编译安装包：

```
# wget http://www.rabbitmq.com/releases/rabbitmq-server/v3.5.4/rabbitmq-server-
generic-unix-3.5.4.tar.gz
```

解压：

```
# tar zxvf rabbitmq-server-generic-unix-3.5.4.tar.gz -C /opt
```

建立软链接：

```
# cd /opt
# ln -s rabbitmq_server-3.5.4 rabbitmq
```

配置环境变量：

```
# vi /etc/profile
```

增加一行：

```
export PATH=$PATH:/opt/rabbitmq/sbin
```

保存后使用下列指令让配置立即生效：

```
# source /etc/profile
```

启用 management plugin：

```
# rabbitmq-plugins enable rabbitmq_management
```

启动 RabbitMQ 服务：

```
# rabbitmq-server -detached
```

5. 管理 RabbitMQ

增加一个管理员用户 admin，密码为 123456。

```
# rabbitmqctl add_user admin 123456
```

将 admin 加入管理组。

```
# rabbitmqctl set_user_tags admin administrator
```

假如你的 Linux 服务器的 IP 地址是 192.168.1.214，在浏览器中输入下列网址打开控制台：

```
http://192.168.1.214:15672/
```

使用 admin 登录，在 Admin 区域中，创建一个在程序中可以使用 RabbitMQ 服务的用户，如 alan，如图 D-1 所示。

创建完成后如图 D-2 所示，这时用户 alan 还没有使用消息通道的权限，显示为 No access。

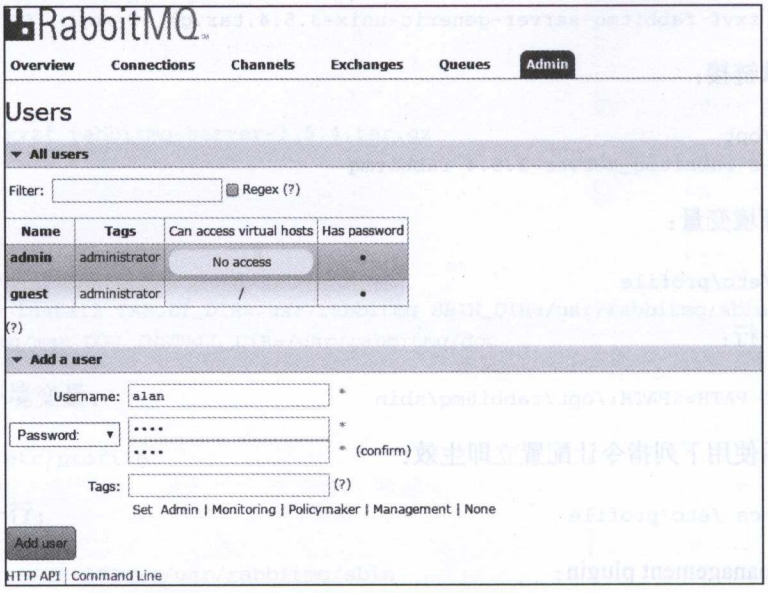


图 D-1 RabbitMQ 管理界面

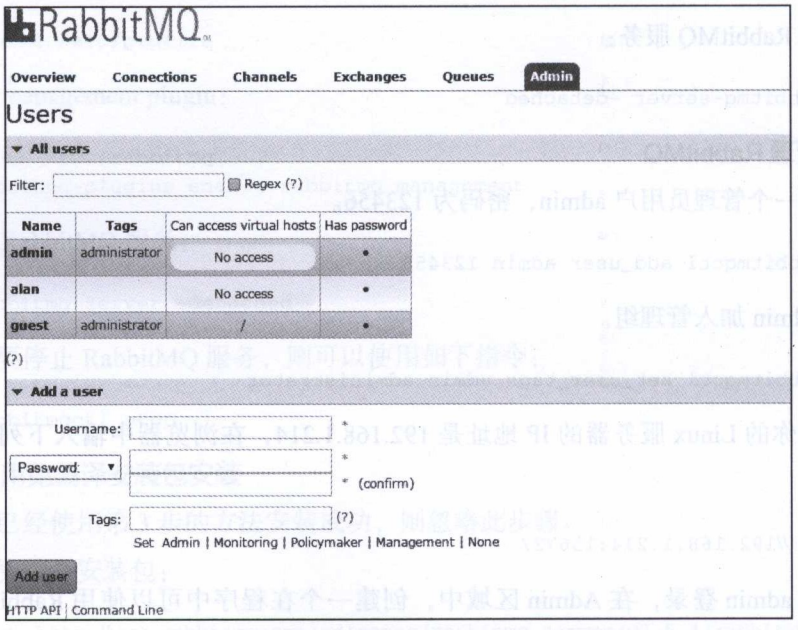


图 D-2 新增加的用户 alan

单击用户 alan，在出现的编辑界面上使用默认选项单击 Set Permission，赋予它读写消息的权限，如图 D-3 所示。

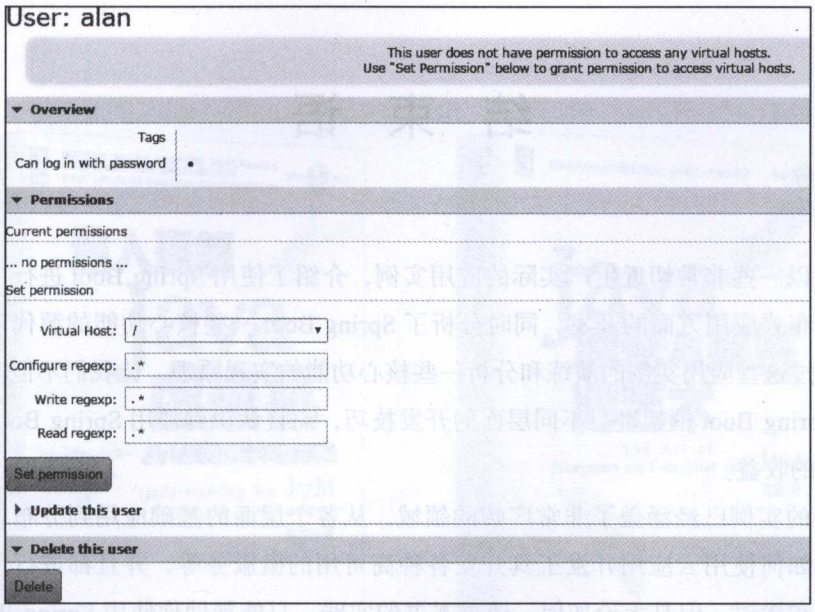


图 D-3　编辑用户权限

现在即可看到用户 alan 已经有了使用消息服务的权限，如图 D-4 所示。这时可以在程序中使用这个用户来连接 RabbitMQ 服务器了。

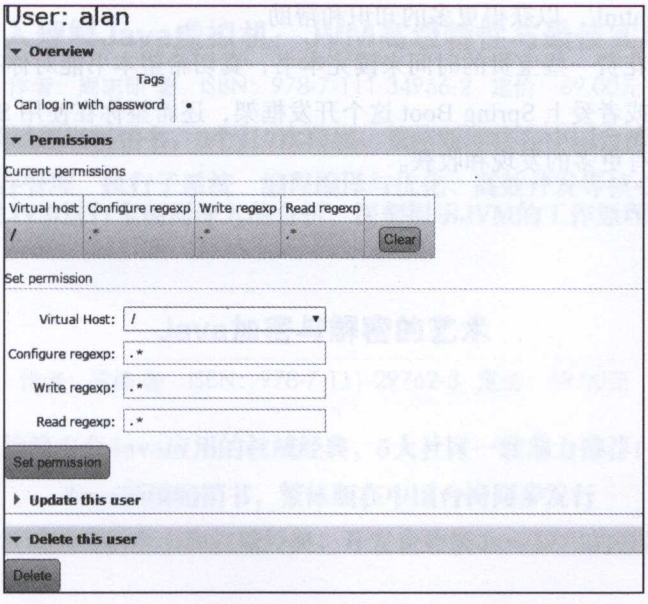


图 D-4　赋予用户 alan 使用消息服务的权限

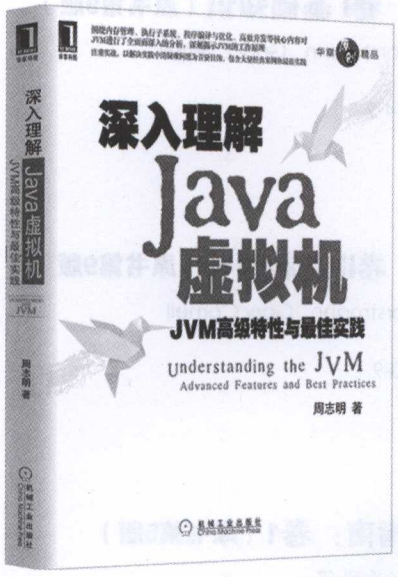
结 束 语

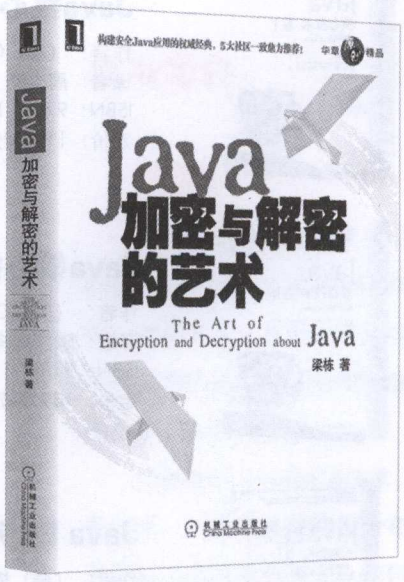
本书以一些非常切近生产实际的应用实例，介绍了使用 Spring Boot 进行一些基础应用和分布式应用方面的开发，同时分析了 Spring Boot 一些核心功能的源代码和实现原理。通过这些应用实例的演练和分析一些核心功能的实现原理，让我们不但掌握了如何使用 Spring Boot 框架进行不同层面的开发技巧，而且认识到使用 Spring Boot 能带来前所未有的收益。

书中的实例已经涵盖了非常广阔的领域，从各个层面的基础应用到分布式管理系统，以及如何使用云应用开发工具开发各种高可用的微服务等，并且都进行了一定的深度挖掘和探讨。但是无论如何，通过本书的实践，只能帮助你使用 Spring Boot 框架进行开发，让你在使用 Spring Boot 的过程中更好地发挥你的长处，却并不能提供更多关于 Spring Boot 的各个方面的参考，而且 Spring Boot 时刻都在发展之中，有关这些方面的内容读者不妨关注 Spring Boot 的官方网站 http://docs.spring.io/spring-boot/docs/current/reference/html/，以获得更多的知识和帮助。

非常高兴你花费一些宝贵的时间来读完本书，真切希望本书能对你有所帮助。如果你能因此而喜欢或者爱上 Spring Boot 这个开发框架，还希望你在使用 Spring Boot 开发框架的过程中，有更多的发现和收获。

推荐阅读





深入理解Java虚拟机：JVM高级特性与最佳实践

作者：周志明 著　ISBN：978-7-111-34966-2　定价：69.00元

Java领域超级畅销书，9个月7次印刷，繁体版即将在中国台湾发行

围绕内存管理、执行子系统、编程编译与优化、高效并发等核心内容
对JVM进行全面而深入的分析，深刻揭示JVM的工作原理

Java加密与解密的艺术

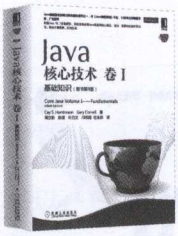
作者：梁栋 著　ISBN：978-7-111-29762-8　定价：69.00元

构建安全Java应用的权威经典，5大社区一致鼎力推荐！

Java领域畅销书，繁体版在中国台湾同步发行

Java安全领域的百科全书和权威经典，开发企业级Java应用的必备参考手册

推荐阅读



Java核心技术：卷I 基础知识（原书第9版）

作者：（美）Cay S. Horstmann　Gary Cornell
译者：周立新 等
ISBN：978-7-111-44514-2
定价：119.00元

Java核心技术：卷II 高级特性（原书第9版）

作者：（美）Cay S. Horstmann　Gary Cornell
译者：陈昊鹏 等
ISBN：978-7-111-44250-9
定价：139.00元

Java EE 7权威指南：卷1（原书第5版）

作者：（美）埃里克·珍兆科 等
译者：苏金国 等
ISBN：978-7-111-49760-8
定价：99.00元

Java EE 7权威指南：卷2（原书第5版）

作者：（美）埃里克·珍兆科 等
译者：苏金国 等
ISBN：978-7-111-49711-0
定价：99.00元

Java应用架构设计：模块化模式与OSGi

作者：（美）Kirk Knoernschild
译者：张卫滨
ISBN：978-7-111-43768-0
定价：69.00元

作者简介

陈韶健（Chen Shaojian）

华阳通信技术总监，有超过15年的IT行业从业经验。从普通的程序开发到系统分析、架构设计，从服务器的组建、维护到系统性能的优化和安全策略实施等，都积累了相当丰富的实践经验。

尤其擅长Java和C#技术，有着深入的实践经验。在数据库使用和分布式应用系统的开发和实施方面也有深入的研究和探讨，并取得了丰硕的成果。

利用Spring Boot进行了大量的项目实践，对其有深入的理解。

封面设计　锡　彬

深入实践
Spring Boot

Deep into Spring Boot

简单易用的Spring Boot，无疑是Java开发初学者的指路明灯，同时也是资深Java开发者的得力助手。快速开发是研发Spring Boot的初衷，这不但是一个开发团队的终生追求，也是一个企业解放生产力、提高生产效率的保障。

Spring Boot的组件化整合规则，完美地整合了云应用开发工具，使其在云计算领域中处于领先地位，为创建高可用和高性能的服务提供了更加简便和快捷的方法。

Spring Boot是从Spring框架发展起来的，所以对于使用Spring框架的庞大用户群体来说，随着Spring Boot的普及使用，将使众多开发者成为它的拥趸。

本书以丰富而又切合生产实际的实例，通过循序渐进的方式以及通俗易懂的语言描述，引领你全面而深入地掌握Spring Boot这一开发框架的使用方法，使你在愉悦地体验一种前所未有的开发实践之后，将所学快速地融入到实际应用之中。

上架指导：计算机\程序设计

ISBN 978-7-111-55088-4

9 787111 550884 >

定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn